NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

PARALLEL PROCESSING WITH TREECLUST

by

I. Taylor McKechnie

September 2017

Thesis Advisor: Samuel E. Buttrey
Second Reader: Lyn R. Whitaker

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704–0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2017	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE
PARALLEL PROCESSING WITH TREECLUST

5. FUNDING NUMBERS

6. AUTHOR(S) I. Taylor McKechnie

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB number ____N/A____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

Clustering data is one of the most common statistical and machine learning techniques for analyzing big data. Clustering can be particularly difficult when the data sets include categorical, missing, or noise variables. The tree clustering algorithm developed by Samuel Buttrey and Lyn Whitaker, as described in the December 2015 issue of The R Journal, seems to provide a solution to these problems, but it requires a large set of overhead computations. This issue is intensified when working with high-dimensional data because the extent of treeClust's overhead computations are based on the dimensions of the data.

High performance computing (HPC) and parallel processing present a solution to this overhead computation burden, but treeClust's existing parallel processing method does not work on the Naval Postgraduate School's HPC, the Hamming Supercomputer (HSC). Furthermore, correctly determining what HPC resources to use can be a difficult task.

In this thesis, we present a new HSC-specific method for parallel processing data using the treeClust R package developed by Buttrey and Whitaker. Based on the results of our experiments, our method approximates the optimal resource HPC request, so that users realize the best run time when using treeClust on the HSC.

14. SUBJECT TERMS tree clusters, parallel processing, high performance computing, big data sets, batch scripting, information systems technology	**15. NUMBER OF PAGES** 131
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540–01-280-5500 Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

PARALLEL PROCESSING WITH TREECLUST

I. Taylor McKechnie
Captain, United States Marine Corps
B.S., Iowa State University, 2010

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

NAVAL POSTGRADUATE SCHOOL
September 2017

Approved by: Samuel E. Buttrey
Thesis Advisor

Lyn R. Whitaker
Second Reader

Patricia A. Jacobs
Chair, Department of Operations Research

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Clustering data is one of the most common statistical and machine learning techniques for analyzing big data. Clustering can be particularly difficult when the data sets include categorical, missing, or noise variables. The tree clustering algorithm developed by Samuel Buttrey and Lyn Whitaker, as described in the December 2015 issue of *The R Journal*, seems to provide a solution to these problems, but it requires a large set of overhead computations. This issue is intensified when working with high-dimensional data because the extent of treeClust's overhead computations are based on the dimensions of the data.

High performance computing (HPC) and parallel processing present a solution to this overhead computation burden, but treeClust's existing parallel processing method does not work on the Naval Postgraduate School's HPC, the Hamming Supercomputer (HSC). Furthermore, correctly determining what HPC resources to use can be a difficult task.

In this thesis, we present a new HSC-specific method for parallel processing data using the treeClust R package developed by Buttrey and Whitaker. Based on the results of our experiments, our method approximates the optimal resource HPC request, so that users realize the best run time when using treeClust on the HSC.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

AWS	Amazon Web Services
CPU	Central Processing Unit
CMDS	Classic Multidimensional Scaling
GB	Gigabyte (1000^3 bytes)
GHz	Gigahertz
GPU	Graphics Processing Unit
HAR	Human Activity Recognition Using Smartphones
HPC	High Performance Computer
HSC	Hamming Supercomputer
INLP	Integer Non-Linear Program
I/O	Input/Output
KB	Kilobyte (1000 bytes)
MB	Megabyte (1000^2 bytes)
MPI	Message Passing Interface
msec	Millisecond
nsec	Nanosecond
RAM	Random Access Memory
SLURM	Simple Linux Utility Resource Manager
TB	Terabyte (1000^4 bytes)
UCI	University of California Irvine

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Cluster analysis is a statistical and machine learning technique used to find structure in a set of measurements. Categorical data, data with missing values, and data with lots of noise regularly present a problem for many established algorithms, and military data sets often include these variable types, making this a relevant problem for military researchers.

The tree clustering algorithm developed by Buttrey and Whitaker (2015, 2016), treeClust, seems to provide a solution to these problems, but it requires a large set of overhead computations. This issue is intensified when working with high-dimensional data because the extent of treeClust's overhead computations are based on the dimensions of the data. High performance computing (HPC) and parallel processing present a solution to this overhead computation burden, but treeClust's existing parallel processing method does not work on the Naval Postgraduate School's HPC platform, the Hamming Supercomputer (HSC). In this thesis, we present a new method for parallel processing data using the treeClust R package developed by Buttrey and Whitaker (2015), on the HSC.

The treeClust algorithm consists of two to three major computations: building trees for each variable, computing dissimilarities between observations—referred to here as computing the dissimilarity function—and clustering the results. This thesis focuses on computing the trees and the dissimilarity between observations. How quickly these computations are performed is dependent on how we use the HSC's resources.

Computers make use of three things to process information: processing units, processing software (referred to here as tasks), and memory. When processing large data sets on an HPC, any algorithm will need to balance its use of these resources in order to reduce run time. In order to use these resources, a program will have to partition its data and spread it out across them. The more resources it uses the more partitioned its data will become, and the more time the program will spend communicating between resources instead of actually processing information. Our method approximates the optimal amount of resources and amount of data partitioning to perform for any particular data set when using treeClust.

Our approach is similar to the existing method but has a few distinct differences. We start out with a parent task and then use shell scripting and system calls to start several child tasks over which we divide the bulk of our work. This is very similar to the method implemented in the R statistical computing environment's parallel package (R Core Team, 2017) in that we are starting many instances of R in order to accomplish our goal. Our method differs from the parallel package in three ways. First, we do not create a socket to communicate back and forth between tasks. In order to communicate between our parent and child tasks, we write to and read from memory. This is expensive but necessary because we do not establish a socket. Second, we give the user the ability to compute the dissimilarity object in parallel. This requires some changes in the way that each tree-based dissimilarity measure is calculated, which we will describe in detail. Finally, we develop a way to approximate the optimal number of tasks to start based on the dimensions of the data set.

In order to evaluate our method, we used two data sets, the Internet Ads data, and the Human Activity Recognition Using Smartphones or HAR data. For any data set, the number of variables (columns) is denoted by *p*, and the number of observations (rows) n. The Internet Ads data has $p = 1558$ columns, and $n = 3279$ rows. The HAR data has $p = 561$ columns, and $n = 10299$ rows (Lichman, 2013a; Lichman, 2013b; Kushmerick, 1998; Anguita, Ghio, Oneto, Parra, & Reyes-Ortiz, 2013).

Our method appears to provide an effective way to perform parallel processing with treeClust on the HSC. For both data sets tested, the total run time oscillates as the number of tasks (and cores) increases. This is shown in Figure 1 and Figure 2. However, the run time realized a generally decreasing trend as the number of tasks used increased, until the $\text{number-of-tasks} \approx 25\% \times p$. We believe the oscillation in run time has to do with the tradeoff between partitioning the data and using more cores. When our method uses more cores, it has to split the data into smaller chunks. These chunks then must be distributed across the HPC, which takes substantial time. If each additional task does not substantially reduce the run time for each task, then the time required to access additional nodes or racks on the HSC becomes a burden on the total run time.

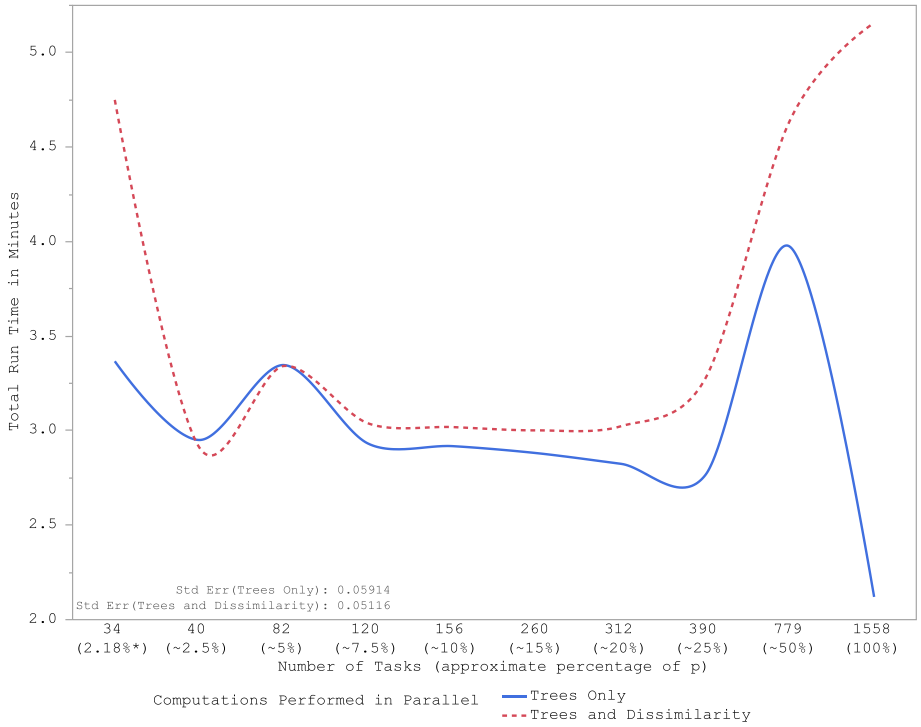


Figure 1. Median Total Run Time: Internet Ads Data Set

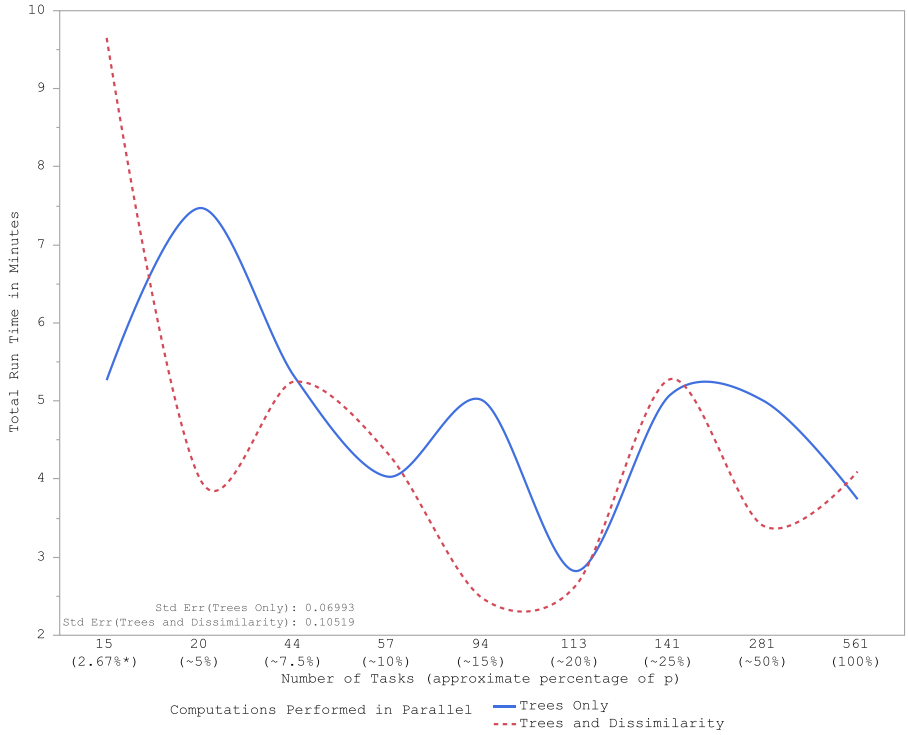


Figure 2. Median Total Run Time: HAR Data Set

Conventional wisdom for processing large data sets states that p tasks should be used to process p variables. For both data sets, when only computing the trees in parallel, p tasks resulted in a reduced run-time; however, the run time for p tasks was not consistently the fastest. When the $\text{number-of-tasks} \approx 20\% \times p$ for the HAR data set (Figure 2), the median run time was faster than the run time for $\text{number-of-tasks} = 100\% \times p$.

Using parallel processing to compute the dissimilarities had mixed results. Based on the data sets used, it appears that as n grows processing the dissimilarities in parallel becomes more beneficial. When n is small the treeClust algorithm is faster when only computing the trees in parallel. This is seen in the HAR data results in Figure 2.

When computing both the trees and the dissimilarities in parallel using the Internet Ads data set (Figure 1), the algorithm realized a drastic increase in run time when p tasks were used. For the HAR data set, the fastest median run time occurred when the $\text{number-of-tasks} \approx 15\% \times p$. Since p tasks only resulted in the fastest median, run time in a quarter of our experiments we believe the optimal number of tasks is a fraction of p. We incorporate these results into our initial resource request, by instantiating a number of tasks between 7.5 and 20% of p.

Our experiments indicate that our specialized approach provides an extensible set of functions that allow future users to take optimal advantage of the HSC with treeClust.

References

Anguita, D., Ghio, A., Oneto, L., Parra, X., & Reyes-Ortiz, J. L. (2013). A public domain dataset for human activity recognition using smartphones. *21st European Symposium on Artificial Neural Networks* (pp. 437–442). Bruges: ESANN.

Buttrey, S. E., & Whitaker, L. R. (2015). treeClust: An R package for tree-based clustering dissimilarities. *The R Journal, 7*(2), 227–236. Retrieved from https://journal.r-project.org/archive/2015/RJ-2015-032/index.html

Buttrey, S. E., & Whitaker, L. R. (2016). *A scale-independent, noise-resistant dissimilarity for tree-based clustering of mixed data.* Monterey, CA: Naval Postgraduate School. Retrieved from http://hdl.handle.net/10945/48615

Kushmerick, N. (1998, July). *Internet Advertisements Data Set*. Retrieved from UCI Machine Learning Repository: https://archie.ics.uci.edu/ml/datasets/Internet+advertisements

Lichman, M. (2013a). *Human Activity Using Smart Phones Data Set*. Retrieved from UCI Machine Learning Repository: http://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones

Lichman, M. (2013b). *Internet Advertisements Data Set*. Retrieved from UCI Machine Learning Repository: https://archive.ics.uci.edu/ml/datasets/internet+advertisements

R Core Team. (2017, March). R: a language and environment for statistical computing. Vienna, Austria: R Foundation for Statistical Computing. Retrieved from https://www.R-project.org

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank Professor Buttrey and Professor Whitaker for allowing me to work with them and to work on their algorithm. It was an honor that I truly enjoyed.

I would also like to thank my family for their constant support and encouragement.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

I wanna go fast!

—Ricky Bobby
Talladega Nights: The Ballad of Ricky Bobby, 2006

A. THESIS PURPOSE

Cluster analysis is a statistical and machine learning technique used to find structure in a set of measurements. It works by grouping observations that share similar aspects (Hartigan, 1975), and is often used when the number of groups in a data set is unknown. Since the data sets are unsupervised, clustering can be very difficult. Making things more difficult, many clustering algorithms are also limited in the kinds of data that they can process. Categorical data, missing values, and data with lots of noise regularly present a problem for many algorithms. Military data sets, such as U.S. Army Recruiting data (Fulton, 2016), often contain many categorical variables, missing values, and noise variables, making this a relevant problem for military researchers.

The tree clustering algorithm developed by Buttrey and Whitaker (2015, 2016), treeClust, seems to provide a solution to many of these problems, but it requires a large set of overhead computations. This issue is intensified when working with high-dimensional data because the extent of treeClust's overhead computations are based on the dimensions of the data. Parallel processing presents a solution to this overhead computation burden, and the use of high performance computing (HPC) can further reduce it.

In this thesis, we present a new method for parallel processing data with the treeClust R package (Buttrey & Whitaker, 2015). Our method is specifically designed for use with the Naval Postgraduate School's (NPS) Hamming Supercomputer (HSC). TreeClust's existing parallel processing method does not work on the HSC, or take full advantage of treeClust's parallelizable computations. In order to get the best

performance, we formulate an integer non-linear program (INLP) to assist future users when determining the best way to use the HSC with their research.

B. THESIS OBJECTIVES

The objectives of this thesis are the following:

- Develop a parallel processing method that allows the computational capabilities of an HPC to be leveraged against a data set with treeClust.
- Take further advantage of the parallelizable computations that make up the treeClust algorithm.
- Develop an extensible set of functions so that others may use this method in future analysis.

C. TREECLUST

TreeClust is implemented as an R package that produces dissimilarities that can be used for clustering data. In Buttrey and Whitaker's algorithm "these dissimilarities arise from a set of classification or regression trees, one with each variable in the data acting in turn as the response, and all others as potential predictors" (Buttrey & Whitaker, 2015, p. 227). The dissimilarities produced by the trees are insensitive to scaling and appear to work well with categorical and mixed data. Additionally, the package can create a new numeric data set whose inter-point distances relate to the treeClust ones. This new data set is significantly smaller and particularly useful when dealing with big data. The process of creating trees and calculating dissimilarities is highly iterative and thus, easily benefits from parallel processing and high performance computing.

D. PARALLEL PROCESSING

1. What is parallel processing?

Parallel processing is the use of a set of processing elements to solve large problems quickly (Almasi & Gottlieb, 1989). A set of processing elements can include multiple central processing units (CPUs), multiple graphical processing units (GPUs), or multiple cores on a CPU or GPU. The computations that make up the problem are spread across the elements to reduce run-time. This is distinctly different from running multiple programs at the same time, known as multiprogramming and multithreading. In

multiprogramming and multithreading, only a single processing element is used, and the computer rapidly switches back and forth between different computations.

2. Why parallel processing?

Why do we not just use a very fast processor and let it handle all of the work? The answer this question is that, modern computers are encroaching on the upper limit of CPU speed (Tanenbaum, 2008). These limits on CPU speed mean that a single processor capable of performing our computations fast enough does not exist, so we turn to the use of many processors at once to solve the problem.

E. HIGH PERFORMANCE COMPUTERS

Conventionally speaking, a HPC consists of thousands or millions of processing elements—CPUs, GPUs, or cores—spread out over several interconnected components. Each element has access to large amounts of volatile and non-volatile memory space, usually in the range of several hundred gigabytes (GBs) of main memory and several thousand terabytes (TBs) of non-volatile memory. The computer's internal network is designed for quick communication between interconnected elements, which makes it much faster than a distributed system. Typically, a HPC runs specially designed software that makes it easy for users to access a large number of processing elements, and large amounts of memory, without an overhead computation burden. HPCs perform well with parallel computations, allowing them to make short work of computations that would take hours or days on a personal computer.

F. ORGANIZATION OF THE STUDY

The remainder of this of thesis is organized as follows. Chapter II is a review of background and literary material relevant to the development of our method. Chapter III is a description of our method and procedures. Chapter IV describes the tests and experiments we perform to evaluate our method. Chapter V provides the results of our experiments. Chapter VI presents our conclusions and possible future work to improve and extend our method.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND AND LITERATURE REVIEW

A. INTRODUCTION

In this chapter, we cover background material related to the four overall subjects of this thesis. Since computer design and capability assumptions significantly affect parallel processing, we begin with a review of relevant computer architecture material. Specifically, Section B covers approaches we will take, and will not take, due to the HSC's design. Section C covers the capabilities and limitations of the R programing language, which we use to implement our method. Finally, Section D reviews the treeClust algorithm, and its R functions.

B. COMPUTER ARCHITECTURE AND THE HAMMING SUPERCOMPUTER

Computer architecture can be broken down into two fields: hardware and software. We begin with hardware.

1. Processing Hardware

CPUs, GPUs, and cores are the pieces of hardware responsible for executing the instructions that make up a program (Abd-El-Barr & El-Rewini, 2005). CPUs and GPUs are the common names for processing units, but often they consist of a varying number of smaller cores. Modern personal laptop and desktop CPUs often contain two to four cores. Currently the HSC has 4130 CPU-based cores, and 600 GPU-based cores (Sharrock & Haferman, 2017). An individual user can access up to 1024 cores at once (Haferman, 2017). GPUs are typically less powerful than CPUs, and require the use of special software for mathematical computations (Blair, 2012). Due to the added software complexity, we will not pursue the use of GPUs in this thesis.

The normal way to measure computer performance is through elapsed time, or the number of seconds it takes a program to execute (Patterson & Hennessy, 2007). This is different from the execution time for specific instructions, which is measured in clock cycles (Patterson & Hennessy, 2007). The length of a clock cycle varies for different

CPUs, but modern processors typically have a clock cycle measured in fractions of a nanosecond. The inverse of a clock cycle, commonly known as clock speed or CPU speed, is given in gigahertz (GHz). For example, a 4GHz CPU has a clock cycle of 0.25 nanoseconds.

The HSC is made up of several different kinds of CPUs, each with different speeds (Haferman, 2017). This means our method will realize different run times based on the CPUs we are allocated. However, limiting ourselves to only the fastest subset CPUs would drastically reduce the number of cores we have access to, also affecting run time. For this reason, we will assume the HSC cores are homogenous from now on.

2. Processing Software

There are four software components used to execute instructions that we care about: processes, threads, sockets, and a scheduler. We begin with processes.

a. Processes (Tasks) and Multiprogramming

Any computer program uses one or more processes to execute its instructions. Simply put, a process is an abstraction of a running program (Tanenbaum, 2008). This abstraction allows a processing unit to multitask by running more than one process at a time. Every process is allocated its own address space in memory where it stores its instructions, and any data it operates on. In order to perform multitasking, the computer switches back and forth between the running processes very quickly. This is known as multiprogramming.

Multiprogramming is a computationally expensive process because it requires re-writing large parts of memory when the computer switches processes (Tanenbaum, 2008). The HSC defaults to running one process per core because of this (SLURM Workload Manager, 2017), and we do not attempt multiprogramming in our method. For grammatical clarity, a process is referred to as a "task" from this point on.

b. Threads and Multithreading (Hyperthreading)

Threads are easiest to think of as a "lightweight" task (Tanenbaum, 2008). They exist inside of a task, running simultaneously (Harris & Harris, 2007). The key difference from a task and a thread is the thread's lack of its own address space. Every thread in a task shares the same memory and processing unit. This is good, because it means that switches between threads happen 10 to 100 times faster than task switches (Marsh, LeBlanc, Markatos, & Scott, 1991), but it can be bad when threads need access to different instructions or different parts of the data. Threads tend to perform best when a large amount of computation and I/O needs to occur, and when the task has access to several cores (Tanenbaum, 2008). Unfortunately, the HSC does not allow multithreading on all of its cores (Haferman, 2017), and because we do not want to reduce the number of cores available to us we will not pursue multithreading in this thesis.

c. Sockets

Sockets are a software interface used to communicate between different tasks or computer nodes (Kurose & Ross, 2010). They can be created dynamically and are not linked to a specific set of hardware. Creating a socket requires a "three-way handshake" between two tasks. This is shown in Figure 1. During the three-way handshake the client task pings the server task on a commonly accessible socket, shown here as the "Welcoming socket," asking it to create a dedicated connection between them. If the server accepts, it opens a new socket between them. This technique is commonly used in many parallel processing methods to allow rapid communication between tasks. We will come back to sockets when we cover how the treeClust package currently implements parallel processing.

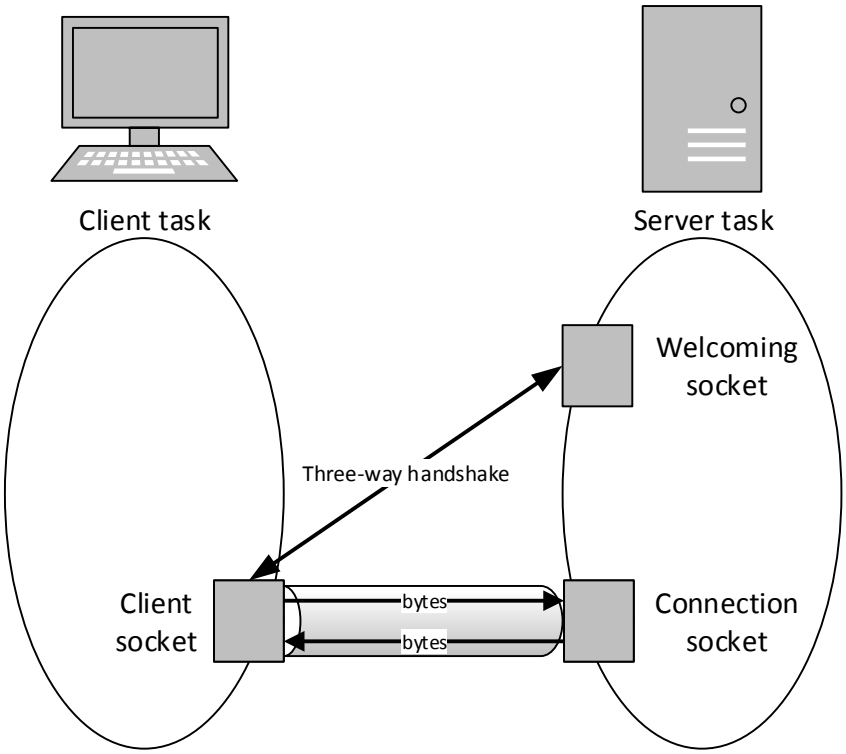


Figure 1. Creating a Socket. Adapted from Kurose and Ross (2010).

d. Scheduler

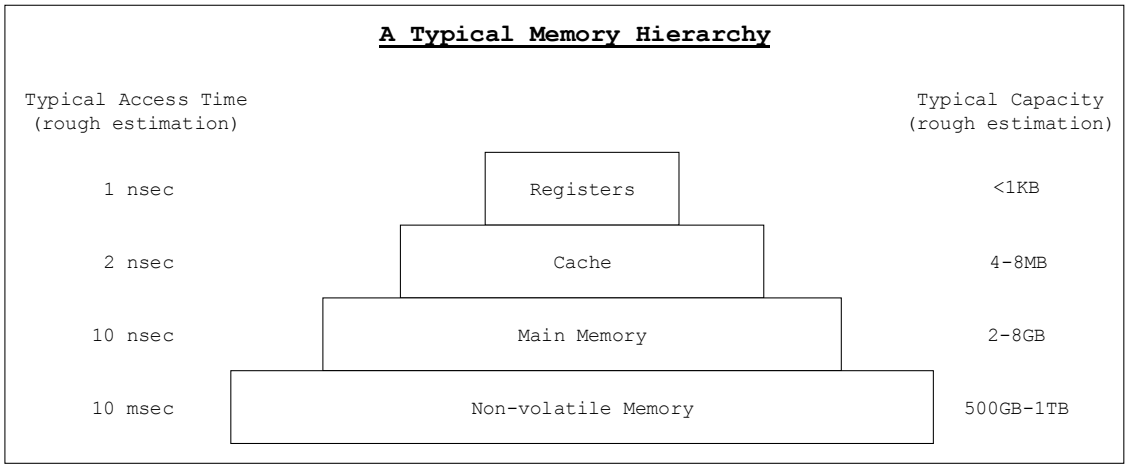
In order to run tasks, computers make use of a CPU manager known as a scheduler. Its focus is to make efficient use of the CPU; and it does so by determining when a task has access to the CPU, and for how long (Tanenbaum, 2008). Since the HSC is a large system made up of many interconnected components, scheduling a task on a particular component and then moving the required instructions and data to the memory for that component, can take a non-trivial amount of time. An example of this is shown in Figure 2. Attempting to predict exactly how long a process will take is subject to large error (Tanenbaum, 2008). This means the run time for any parallel processing method made up of multiple tasks will vary, and the method will need to be thoroughly tested to determine a median run time.

The bottom three tasks started executing a full 21 seconds before the top 17 tasks. This occurs because the data for each task is sent to another node, which requires a non-trivial amount of time. The `JOBID` column lists the name of each task; the `TIME` column lists how long each task has been running; the `NODELIST(REASON)` column lists the node the task was sent to by the scheduler.

Figure 2. Example Output from the HSC Task Scheduler

3. Memory

All forms of processing require reading from memory. In a computer, memory is arranged in a hierarchy, shown in Figure 3. We are concerned with the speed at which we can access the instructions and data we have stored in memory, because longer wait times equal slower run times.



The typical capacity estimates have been updated from Tanenbaum's work to reflect current norms. Typical access times remain approximately the same.

Figure 3. A Typical Memory Hierarchy. Adapted from Tanenbaum (2008).

a. Registers and Caches

This is memory located directly on the CPU; it has the fastest access time, but very low capacity. Registers are primarily used to store instructions and key variables (Abd-El-Barr & El-Rewini, 2005). Caches are used to hold pieces of a task that will not fit into the registry. Any instructions or data that cannot be stored in the registry or caches are kept in main memory or non-volatile memory. As Figure 3 shows, when a task has to execute a fetch to main memory it realizes an order of magnitude increase in access time. We would like to limit the number of fetches to main memory as much as possible.

For every task switch all registers and caches must be re-written, and the new instructions and data must be loaded from main memory (Tanenbaum, 2008). As we mentioned before this is an expensive action which we would prefer not to do. This reinforces our decision to avoid multiprogramming, and to use the HSC cores on their default setting.

b. Main Memory and Virtual Memory

Main memory, commonly known as random access memory (RAM), is the primary storage for a running process (Shiva, 2008). While slow in comparison to registers and caches, it is several orders of magnitude faster than non-volatile memory.

The HSC contains several thousand GBs of RAM, and each node has access to 35–500 GBs (Sharrock & Haferman, 2017). When the available RAM runs out, the computer will resort to using Virtual Memory, which is a software technique that allows non-volatile memory to be used like RAM (Patterson & Hennessy, 2007). When the computer begins storing objects for a task in Virtual Memory its access time to that data is increased substantially, and the task's run time increases.

c. Non-volatile Memory

Non-volatile memory constitutes the largest and slowest form of memory a computer has access to. Non-volatile memory has many forms, but hard drives and solid state drives make up the vast majority of what is used. The primary purpose of non-volatile memory is for storage, and overflow main memory. The HSC has several TBs of non-volatile memory in the form of solid state drives (Haferman, 2017).

4. Hardware Organization and Storage

The way computer hardware is organized and stored also affects the run time of any computer program using parallel processing. For the HSC in particular each circuit board – or node – contains 6–64 cores (Sharrock & Haferman, 2017). Accessing cores on the same node occurs relatively quickly, but accessing cores contained on another node can require considerable additional time (see Figure 2). Since the HSC's nodes contain a maximum of 64 cores, our method will use several nodes to process large data sets. This also presents our method with a data partitioning vs processing power decision. Attempting to use too many cores will result in a run-time delay because of the communication time between nodes. Using too few nodes will leave us with a lack of processing power. Our analysis attempts to determine a good number of cores based on the dimensions of the data set.

5. Running Programs on the Hamming Supercomputer

Most modern computers have some form of command environment, or language, which users use to execute tasks or run programs. For example, Windows uses cmd.exe and Macintosh systems use terminal. The HSC is no different; and it uses the SLURM

Workload Manager and Bash command language to perform similar functions (Haferman, 2016). SLURM stands for Simple Linux Utility Resource Manager, and it is used specifically to run "jobs" on the HSC (SLURM Workload Manager, 2017). Jobs are used to execute programs and run tasks. In order to start a job, users must make an initial resource request to be allocated a portion of the HSCs cores and memory. After SLURM grants the user's request, it will begin to either execute tasks or allow the user to run programs, such as R. Our methodology uses the Bash shell command language to communicate with SLURM and run many jobs simultaneously by creating a batch file.

Batch files—also known as batch scripts—are used to execute large numbers of commands repeatedly from the command line (Horstmann, 2008). Running batch files, or shell scripting, on the HSC is done using several specific SLURM commands and variables, which we will cover in our methodology.

C. R, CAPABILITIES AND LIMITATIONS

Our method is implemented in the R programming language, and like every computer program, R has limitations. R's ability to access memory is limited by what is physically available, and by the operating system. This is a non-trivial problem when dealing with big data sets. A common R error when processing big data is "`cannot allocate vector of size X`." This indicates that R has used all available memory and cannot build or hold another object. This can also occur when the operating system does not release previously used memory, which is a known problem for R on Linux machines (R Core Team, 2017). The HSCs nodes have between 35–500 GB of RAM (Sharrock & Haferman, 2017), but this equates to only 3.7-12GB per core. If a task is assigned too much data, it could easily exceed its memory allocation. Big data processing methods often require multiple times this amount of memory to execute their computations. The HSC's command environment enables users to request a specified amount of RAM for each task, and it does not place an upper bound on the amount any individual task can have, but this does not mean a task can exceed the amount of physically available RAM. Our method establishes a constraint in order to ensure each task has access to enough memory.

D. TREECLUST

> The idea of tree-based clustering stems from this premise: objects that are similar tend to land in the same leaves of classification or regression trees. In a clustering problem there is no response variable, so we construct a tree for each variable in turn, using it as the response and all others are potential predictors.
>
> —Samuel E. Buttrey and Lyn R. Whitaker
> "treeClust: An R Package for Tree-Based Clustering Dissimilarities," 2015

The treeClust algorithm consists of two to three major computations: building trees for each variable, computing dissimilarity between observations— known as inter-point dissimilarity – and clustering the results. It is up to the users to determine if the algorithm should perform clustering, and we will not focus on it in this thesis. The computation sequence of the algorithm is included in Figure 4.

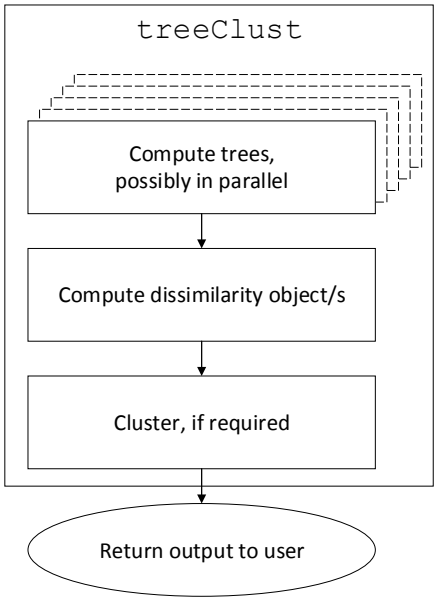


Figure 4. Computation Sequence: treeClust

The R implementation of treeClust consists of several functions used to perform these computations. Of particular interest to us are the functions `treeClust()`, `tcdist()`, and `tcnewdata()`.

1. The treeClust() Function

The `treeClust()` function takes many arguments. The arguments include a rectangular data set (given as an R data frame), the choice of a tree-based dissimilarity measure, a clustering algorithm, the number of clusters, and a set of column names or numbers specifying which variables to build trees for, and a control object—`treeClust.control()`—which allows the user to specify what should be returned by the function (Buttrey & Whitaker, 2015). For this thesis, we are primarily concerned with the data set, the tree-based dissimilarity measure, and the column numbers. We now move on to how the trees are computed.

2. Computing Trees

Trees are built using a modified version of the `rpart()` function from R's rpart package (Therneau, Atkinson, & Ripley, 2015). This is the most computational intense process of the treeClust algorithm due to how many calculations must be performed, the size of the data, and the size of the output from each tree. Each tree outputs four objects:

(1) The tree itself.

(2) A number indicating the number of leaves in the tree.

(3) A vector of length n, detailing the leaf membership for each observation.

(4) A measure of tree quality known as the deviance ratio.

As Figure 4 shows, trees can be built sequentially or in parallel if the user desires. Since a standard call to `treeClust()` requires building a tree for each variable in the data set, this computation done sequentially can take a very long time, so parallel processing is preferred. We will discuss the existing parallel processing method and its limitations at the end of the chapter. We note that after the trees are built, they are pruned to their optimal size – the size for which their cross-validated deviance is the smallest – to avoid overfitting. Any tree that is pruned all the way to the root is discarded (Buttrey & Whitaker, 2015). We now move on to how the deviance ratio is computed, and explain the tree-based dissimilarity measures.

b. Tree-based Dissimilarity

There are four tree-based dissimilarity measures. We will cover each one and its computations in detail.

(1) Tree Distance One, d_1

The most straightforward dissimilarity measure, d_1, shown in Equations (2.1) and (2.2), is given by the proportion of trees in which observations i and j fall in different leaves. Let $L_t(i)$ record the leaf of tree t into which observation i falls. Equation (2.1) gives the dissimilarity between observations i and j, $d_1^t(i, j)$, for tree t. If observations i and j fall in the same leaf of tree t, they have a dissimilarity value of zero for that tree, and one otherwise (Buttrey & Whitaker, 2016).

$$d_1^t(i, j) = \begin{cases} 0 \text{ if } L_t(i) = L_t(j) \\ 1 \text{ if } L_t(i) \neq L_t(j) \end{cases} \tag{2.1}$$

The dissimilarity between observations i and j for the entire data set is then computed by averaging the tree dissimilarities, where T is the number of trees in the data set.

$$d_1(i, j) = \frac{\sum_{t=1}^{T} d_1^t(i, j)}{T} \tag{2.2}$$

The dissimilarities between all pairs of observations in the data set can be averaged in an $n \times n$ dissimilarity matrix. Computation of this matrix is easily parallelizable, because the dissimilarity for each pair of observations can be computed by tree, and then the tree dissimilarities can be summed after all of the trees have been computed.

(2) Tree Distance Two, d_2

The second measure of dissimilarity, d_2, differs from d_1 in that it is based on a measure of tree quality. Tree quality is based on ΔD_t, the proportion of deviance "explained" by the tree. The contribution of each node to the deviance is "the usual sum

of squares from the node mean for a numeric response, and the multinomial log-likelihood for a categorical one“ (Buttrey & Whitaker, 2015, p. 227). For each tree, t, let D_t^{root} be the root node deviance, and D_t^{leaves} be the sum of the l_t leaf node deviances, where l_t is the number of leaves for tree *t*. Then ΔD_t is defined as $\Delta D_t = \frac{D_t^{root} - D_t^{leaves}}{D_t^{root}}$.

Equation (2.3) gives the d_2 tree-dissimilarity between pairs of observations i, j. Like d_1^t for a particular tree, if two observations fall in the same leaf they have a dissimilarity of zero. If they fall in different leaves, their value is tree quality ΔD_t scaled by the maximum tree quality over all trees.

$$d_2^t(i, j) = \begin{cases} 0 & \text{if } L_t(i) = L_t(j) \\ \dfrac{\Delta D_t}{\max\limits_k (\Delta D_k)} & \text{if } L_t(i) \neq L_t(j) \end{cases} \tag{2.3}$$

The d_2 dissimilarity between pairs of observations over the whole set of trees is then computed as an average in the same way as d_1. Equation (2.3) can be computed in parallel by waiting to scale by the maximum tree quality until all trees have been computed.

(3) Tree Distance Three, d_3

Dissimilarity measure three, d_3, no longer assumes that all leaves are equally different from one another. Here, for each tree *t* and pair of observations *i* and *j* define $\Delta D_t(i, j)$ to be the deviance of observations *i* and *j*'s nearest parent node subtracted from the deviance at the tree's root (Lynch, 2014). The d_3 tree dissimilarity between observations *i* and *j* is then defined as Equation (2.4).

$$d_3^t(i, j) = \begin{cases} 0 & \text{if } L_t(i) = L_t(j) \\ \dfrac{\Delta D_t(i, j)}{\Delta D_t} & \text{if } L_t(i) \neq L_t(j) \end{cases} \tag{2.4}$$

The dissimilarity between observations over the whole set of trees is then computed as an average in the same way as d_1 and d_2.

(4) Tree Distance Four, d_4

Dissimilarity measure four is a combination of measures two and three. Like d_3, d_4 accounts for how far apart leaves in a tree are and like d_2, each tree's contribution is weighted by its tree's quality (Buttrey & Whitaker, 2015). Specifically, the tree *t* dissimilarity between observations *i* and *j* is given by Equation (2.5).

$$d_4^t(i,\ j) = \begin{cases} 0 & \text{if } L_t(i) = L_t(j) \\ \left(\dfrac{\Delta D_t(i,j)}{\Delta D_t}\right) \times \left(\dfrac{\Delta D_t}{\max\limits_k (\Delta D_k)}\right) & \text{if } L_t(i) \neq L_t(j) \end{cases} \tag{2.5}$$

Like d_2, d_4 can be computed in parallel by waiting to scale tree-dissimilarities until all trees have been computed. Now we will move on to how the output from the trees is stored.

c. Output Combination

The output from each tree is combined into three objects, which are either returned to the user or used for calculating dissimilarities. An understanding of these objects is necessary because our method uses smaller portions of these objects for calculating dissimilarities than the sequential execution of the `treeClust()` function does. The output from each tree is combined into the following:

(1) The number of leaves, l_t, and deviance ratio for each tree are maintained in a `results` matrix, with one row per tree.

(2) The vector of leaf membership for each tree is maintained in a matrix, `leaf.matrix`, with one column for each tree. The dimension of the final matrix is $n \times T$, where T is the number of trees remaining after pruning.

(3) The actual trees themselves are stored in a list of length T, with one tree per element.

Once these objects have been constructed, inter-point dissimilarities, using one of the four dissimilarities may be computed.

3. Dissimilarity Functions

The R function that is used to compute dissimilarities is `tcdist()`. A second function `tcnewdata()` may be used along with or in place of `tcdist()`. Both are based on tree output and the specified dissimilarity measure, but they differ from one another in their approach.

a. The tcdist() Function

Tree cluster distance, `tcdist()`, computes the lower triangular portion of the $n \times n$ dissimilarity matrix (the matrix containing dissimilarities for all pairs of observations). For d_1, and d_2 this is done using the `daisy()` function from the `cluster` package (Maechler, et al., 2017). The `daisy()` function takes `leaf.matrix` – where each row is an observation and the columns are the T categorical leaf-membership variables – and computes the dissimilarity between its rows. Weights are assigned at run time using the `results` matrix. For d_1 the weight is simply one. For d_2 the weight is the tree quality measure used in Equation (2.3). This results in an object of class `dissimilarity`, which is a vector of length $n(n-1)/2$. A vector is used to save storage space (Maechler, et al., 2017).

For d_3 and d_4, the weights are assigned in similar fashion to d_1 and d_2, but the computation of the dissimilarity matrix requires the use of another treeClust package

function, `d3.dist()`. The function `d3.dist()` "establishes all the pairwise distances between the leaves in a particular tree, and then computes the pairwise dissimilarities among all observations by extracting the relevant leaf-wise distances" (Buttrey and Whitaker, 2015, p. 230). This computation must be done for each tree. Since a matrix of dissimilarities must be computed for each tree, and it is currently done sequentially, this takes a long time (several seconds per tree). The computation of each tree's dissimilarity matrix can be broken up to spread the computation load. Then any scaling based on the full set of trees can be done when the results of each tree are combined. We provide further detail on how this is done in Chapter III.

b. The tcnewdata() Function

The `tcnewdata()` function avoids computing and storing all $n(n-1)/2$ pairwise distances. A matrix `newdata`, with n rows, is created by producing "a new data set in which the inter-point distances are similar to the ones computed by the `treeClust()` function, but arrayed as a numeric data set" (Buttrey and Whitaker, 2015, p. 233).

For d_1 and d_2, a set of $l_t \left(t = 1,...,T\right)$ binary variables is created for each tree, where l_t represents the number of leaves in tree t, and a one indicates leaf membership for each observation. "The resulting data set has Manhattan (sum of absolute values) distances that are exact multiples of the Gower dissimilarities we would have constructed from the matrix of leaf membership" (Buttrey & Whitaker, 2015, p. 233). The new data set has dimensions $n \times L$, L being the total number of unique leaves across all of the trees. This data set can be much smaller than the set of pairwise dissimilarities. d_2's weight scaling is done in a similar fashion to `tcdist()`. Additionally, because each tree independently donates its own set of unique leaves, the `tcnewdata()` function is fully parallelizable. Scaling by the maximum deviance ratio—required for d_2—is simply delayed until all trees, and newdata portions, have been computed.

For distance metrics d_3 and d_4, tree t again donates l_t variables, one per leaf, but a particular tree results in a set of $l_t(l_t-1)/2$ distances between pairs of leaves. An $l_t \times l_t$

symmetric matrix is constructed for each tree, where the $(i, j)th$ entry is the distance between leaves i and j. Classical multidimensional scaling (CMDS) implemented by the `cmdscale()` function, part of base R, is then used to produce a $l_t \times (l_t - 1)$ matrix of coordinates. The scaling for d_4 is done in the same manner as `tcdist()`, and can be delayed until all trees and newdata portions have been computed, enabling parallel processing.

4. Existing Parallel Processing Method

TreeClust can currently implement the function `clusterApplyLB()` from R's parallel package (R Core Team, 2017) to build trees in parallel if the user requests to do so. The function uses an "embarrassingly parallel" method, meaning that it works with computations than can be easily divided into independent parts (van Engelen, 2017). The number of chunks that can be processed at any time is limited by the number of cores requested by the user. To process each chunk the function starts a new R task (known as a child task), and each task is run on a separate core. The function then makes use of a load balancing technique to send each divisible chunk of the program to a task once the task is finished with its previous chunk. This cycle repeats until all chunks have been processed (Tierney, Rossini, Li, & Sevcikova, 2016). Unfortunately, this package does not work on the HSC.

In order to start each task and communicate between them, `clusterApplyLB()` must be passed a socket cluster of cores (Tierney, Rossini, Li, & Sevcikova, 2016). This is computed using the `makePSOCKcluster()` function from the same package. A fatal error occurs here because the HSC's command environment does not allow `makePSOCKcluster()` to create the socket without using additional software.

To complete the three-way handshake and establish the socket, `makePSOCKcluster()` must make use of message passing interface (MPI) software (Lockwood, 2017). MPI utilization requires the installation of additional R packages and the use of other HPC command environment software, making it beyond the scope of this thesis. Without MPI software to establish the socket, `makePSOCKcluster()` fails and

`clusterApplyLB()` executes using only the parent task. This means that `treeClust()` is forced to compute in serial mode, leaving the overhead computation burden in place.

5. Alternative Parallel Processing R Packages

R's CRAN repository contains more than 80 different packages to perform high performance computing and parallel processing (Eddelbuettel, 2017). Reviewing each package and finding the best one is a time-intensive effort, requiring an in-depth knowledge of many additional software packages or computer science techniques. In this thesis, we present a method to perform parallel processing on the HSC using functions that are included in all basic installations of R.

THIS PAGE INTENTIONALLY LEFT BLANK

III. METHODOLOGY

A. INTRODUCTION

Our approach is similar to the existing method, but has a few distinct differences. We start out with a parent task and then use shell scripting and system calls to start several child tasks over which we divide the bulk of our work. This is very similar to the method implemented in the parallel package, in that we are starting many instances of R in order to accomplish our goal. Our method differs from the parallel package in three ways. First, we do not create a socket to communicate back and forth between tasks. In order to communicate between our parent and child tasks we write to and read from memory. This is expensive, but necessary because we do not establish a socket. Second, we give the user the ability to compute the dissimilarity object in parallel. This requires some changes in the way that each tree-based dissimilarity measure is calculated, which we will describe in detail. Finally, we develop an INLP to approximate the optimal number of tasks to start based on the dimensions of the data set.

B. NEW AND MODIFIED TREECLUST R FUNCTIONS

In this section, we describe the R functions used to implement our method. First we explain the purpose and sequence of each function, and then we describe them in detail. We place particular emphasis on the new functions `tch.childTaskRunner()`, `tch.combiner()`, and the modifications to existing functions `tcdist()` and `tcnewdata()`. We use the tch prefix—which stands for treeClust High Performance—to differentiate our new and modified functions from the standard functions.

1. Function Sequence

Our method is implemented through three new functions and modifications to three existing treeClust functions. Our first function is a modification of the function `treeClust()`, which we call `tch.treeClust()`. Figure 5 contains an overview of this function. This function serves a parent, or main, function from which all others are called. Our second function, `tch.determineSettings()`, uses our INLP to approximate the

optimal number of tasks to start based on the dimensions of the data set, and some HSC-specific constraints. Our second, and third, functions start child tasks, and then combine the output from those child tasks.

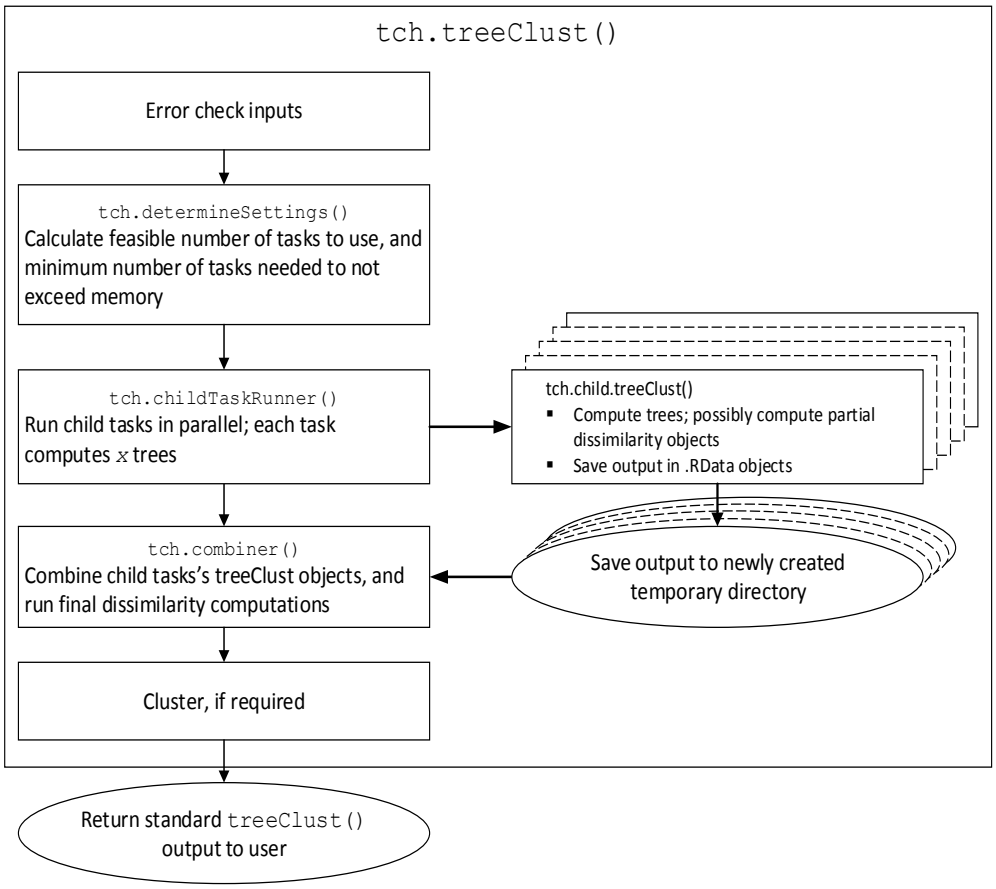


Figure 5. Computation Sequence: `tch.treeClust()`

2. The tch.treeClust() Function

The `tch.treeClust()` function takes all standard `treeClust()` arguments, with the exception of the data frame. Since we do not communicate between tasks using sockets, we cannot actively pass bytes back and forth. Instead, we require the user to submit a network path to the location of the data frame, with the name included. The path is given to each child task, which use the `attach()` function to access the data set when it begins.

In addition to modifying how the data set is acquired, we add six new arguments. The first two arguments, `parallel.dists` and `parallel.newdata`, allow the user to indicate if they would like to use parallel processing to compute respectively the dissimilarity object or the new data object. The third argument is an option to remove the temporary files the function creates during execution. These files are stored in multiple directories created in the working directory using the function `tch.createTempDirs()`. They can be useful for error checking if the algorithm fails. The fourth argument is the user's HSC user name. The user name is needed by `tch.childTaskRunner()` in order to determine when all child tasks have completed. The user name must be passed into `tch.treeClust()` by the user because there is no easy, consistent way to get it from the system. The final two arguments indicate the dimensions of the data set. The dimensions are required because the parent task does not have a working copy of the data which it can query for the information. The dimensions are used by `tch.determineSettings()` which we describe next.

3. The tch.determineSettings() Function

This function takes the dimensions of the data set and approximates the optimal number of child tasks over which to spread the work. To do this we implement the integer non-linear program shown in Equations (3.1) to (3.5), where *R*, *MpC*, *BpO*, *Rlb*, *Rub* are defined in this section. An explanation of each constraint and the objective function follows. Let $R = ceiling\left(\frac{p}{x}\right)$, then

$$\underset{x\in\{1,2,\ldots,p\}}{\text{maximize}} \; R \quad \text{subject to:} \tag{3.1}$$

$$x \le floor\left(\frac{MpC\times 1000^2}{p\times n\times BpO}\right) \tag{3.2}$$

$$x(N-1) < p \tag{3.3}$$

$$R > ceiling(Rlb\times p) \tag{3.4}$$

$$R \le ceiling(Rub\times p) \tag{3.5}$$

First, x represents the number of trees each task will build, where x is an integer between one and p. The ceiling of the ratio p to x gives the number of tasks we use, R. A fractional result indicates the number of variables left for the final task is less than x, but greater than one, so we have to start a task in order to build the remaining tree(s). We can start only an integer number of tasks, so we take the ceiling function of the ratio to get the final number of tasks to use.

The first constraint, Equation (3.2), is a memory-based constraint. In order to avoid memory overflow for any particular task, we estimate the amount of memory needed to store the data set and then calculate how many times it will fit into the RAM allocated to a single core. This is based on the number of bytes used to store each observation (*BpO*), and the amount of memory allocated to each core (*MpC*). This is necessary because each tree must consider the entire data set, and the operating system cannot release previously used memory to build new trees. As we mentioned in Chapter II, this is a known problem for R on Linux machines, like the HSC (R Core Team, 2017).

Equation (3.3) specifies that the final variable included in the penultimate task must be strictly less than the total number of variables in the data set. This ensures two things, first, that the final task has at least one variable, and second that our tasks account for every variable.

Constraints three and four—Equations (3.4) and (3.5)—are heuristic based bounds on the number of tasks to use. *Rub* (percentage of p, upper bound) and *Rlb* (percentage of p, lower bound), are upper and lower bounds which we will determine through testing. Initially *Rub* and *Rlb* are one and zero, respectively. These bounds affect the value of x and are designed to find the optimal number of tasks to use based on the dimensions of the data set.

In the objective function, we assume that processing smaller chunks of data is faster, and therefore we maximize the number of tasks used.

In order to implement this formulation, the parameters *MpC*, *BpO*, *Rlb*, *Rub* are added to the `treeClust.control()` object as four additional arguments with specific default values. *MpC* is assigned a default value of 3700, because that is the minimum

amount of RAM per core on the HSC (Sharrock & Haferman, 2017). We initially assume that the observations of a user's data set are numeric, therefore *BpO* is assigned a default value of eight, because most numbers in R have a size of eight bytes (Grolemund, 2014). *Rlb* and *Rub* receive their heuristic based values by default (these values will be covered in detail in Chapter V).

To solve our INLP we use an iterative search beginning with $x = 1$, and stop once a feasible number of tasks is found. The return value for `tch.determineSetting()` contains two variables: the number of tasks to use (the result of the objective function), and *x* (the number of variables used by each task). These two variables are required arguments for `tch.childTaskRunner()`, which we describe next.

4. The tch.childTaskRunner() Function

The purpose of `tch.childTaskRunner()` is to initiate a SLURM batch job that starts several new instances of R. Each one of these instances runs `treeClust()` on a specific range of the data sets variables. Figure 6 contains a breakdown of the function.

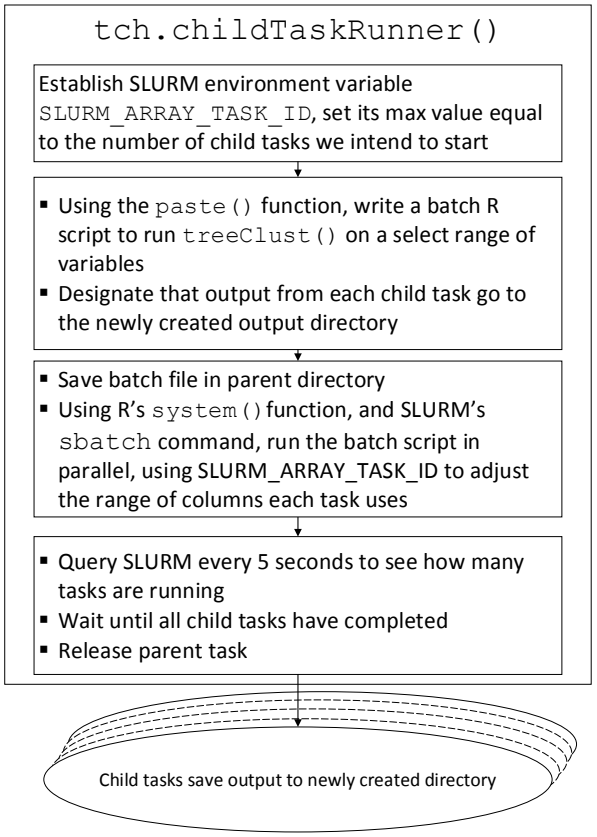


Figure 6. Computation Sequence: `tch.childTaskRunner()`

To correctly initiate a batch job that is specific to the user's inputs, the function takes several arguments. First among these is the path to the data set, and the data set's name. Regular expressions are used to separate the name of the data set from the path, and save it. The function then calls a sub-function, `tch.createTempDirs()`, to create a new directory—named after the data set, and the current system time—where log files from each child task are stored, and another directory where the `treeClust()` output from each child task is stored.

This file management is necessary for two reasons. First, because some trees will be pruned all the way to the root, they will not result in output and no output file will be produced. Because of this, we need a way to determine how many tasks created output. Storing the output from each child task in a single directory allows us to get the name of each file, and then iterate over them with a loop, thus alleviating the need for communication between the child tasks and the parent task. Second, because we do not

want to overwhelm the user's working directory with hundreds or thousands of new files, creating a single directory for storage of these temporary files allows us to perform easy clean up once our computations are complete.

The second and third arguments to `tch.childTaskRunner()` are the number of child tasks to start, and the number of variables from the data set each child task will use. The remaining arguments specify how the dissimilarity object is to be computed – distance measure, dissimilarity computed in parallel or not, clustering algorithm to use – and what objects should be returned. These arguments are used to build a specific R batch script, shown in Figure 7, using the `paste()` function. This batch script also contains a reference to a specific SLURM environment variable, `SLURM_ARRAY_TASK_ID`, which allows the batch script to be iterated over like a loop, by the bash command environment in which R runs. After the batch script is built, it is then saved to the current working directory and called using the SLURM command `sbatch`. This command allows a batch script to be called any number of times. In our case, we set the SLURM environment variable we just mentioned, `SLURM_ARRAY_TASK_ID`, to be equal to the range 1 to *R*, and use it to index over our batch script, starting a new R task with each iteration. Figure 7 shows how `SLURM_ARRAY_TASK_ID` iterates; it changes the column range argument to `treeClust()`, forcing each new task to build a different subset of the trees.

```
# Write the batch file - childBatch.txt - from the given
# inputs to treeClust, the computed number of tasks to
# start obtained from tch.determineSettings(), and the
# number of columns in the data set.
childBatch = paste0(
'#!/usr/bin/Rscript',                                              ' \n',
'library(tree)',                                                   ' \n',
'library(treeClust)',                                            ' \n\n',

'source(\'', new.dirs$org.dir,'tch.SourceFile.R\')',             ' \n\n',

'num = as.numeric (Sys.getenv (\'SLURM_ARRAY_TASK_ID\'))',       ' \n\n',

'attach(\'', new.dirs$org.dir, '/', dfx.name,
       '.RData\', pos=2)',                                       ' \n\n',

'start.col = (num-1) * ', x,' + 1',                                ' \n',
'end.col = num * ', x,                                             ' \n',
'if (end.col >', p,') end.col =', p,                             ' \n\n',

'out = tch.child.treeClust(dfx = ', dfx.name,                       ', \n',
'       d.num = ', d.num,                                           ', \n',
'       col.range = start.col:end.col',                             ', \n',
'       final.algorithm = \'', final.algorithm,'\'',                ', \n',
'       k = ', k,                                                   ', \n',
'       control = tch.treeClust.control(
                  return.trees = ',     tree.return,                ', \n',
'                 serule = ',           control$serule,             ', \n',
'                 DevRatThreshold = ',  control$DevRatThreshold,')', ', \n',
'       parallel.dists = ',   control$parallel.dists,               ', \n',
'       parallel.newdata = ', control$parallel.newdata,             ', \n',
'       verbose = TRUE)',                                         ' \n\n',

'save (out, file = paste0(\'',new.dirs$child.dir,
      'out\' , num,  \'.RData\'))',                                  ' \n')
##end childBatch

# Now write childBatch to a txt file, and send SLURM
# commands to execute it using the system() function
#
# Write the new batch script to a callable file
fileConn = file("childBatch.txt")  #create file connection
writeLines(childBatch, fileConn)   #write childBatch to the file
close(fileConn)                    #close connection

#move into the log directory, so that log files save there
setwd(new.dirs$log.dir)

#make batch file runnable
d2u.cmd = paste0('dos2unix ',new.dirs$run.dir,'childBatch.txt')
system(d2u.cmd)

#change permissions to executable
chmod.cmd = paste0('chmod +x ',new.dirs$run.dir,'childBatch.txt')
system(chmod.cmd)

#set up run command, and run the batch file. This instantiates all child tasks
run.cmd = paste0('sbatch --mem-per-cpu=', control$MpC,' --array=1-', R,
                 ' ',new.dirs$run.dir,'childBatch.txt')
system(run.cmd)
```

This figure shows the creation of the R batch script, and then the creation and execution of SLURM commands. Comments are represented in grey; active commands, variables, and functions are represented in black; the batch script and SLURM commands being generated are in blue.

Figure 7. Batch File Creation and Execution

Every child task is required to save the `results` matrix so that it can be used for the final dissimilarity calculations. `leaf.matrix` is returned if the user did not request to compute the dissimilarity object in parallel, or if the user requests that it be returned. The trees are returned if the user asks for them, or if the user did not specify to compute the dissimilarity object in parallel, and chooses a distance measure of d_3 or d_4.

As Figure 6 shows after the child tasks are instantiated, the function queries the command environment every five seconds to see if they have completed. Once all child tasks have completed, the function releases the parent task back to `tch.treeClust()` in order to call `tch.combiner()`, which we cover next.

5. The tch.combiner() Function

As its name implies, the purpose of the combiner function is to combine the outputs of each child task. The combiner function sequence is shown in Figure 8.

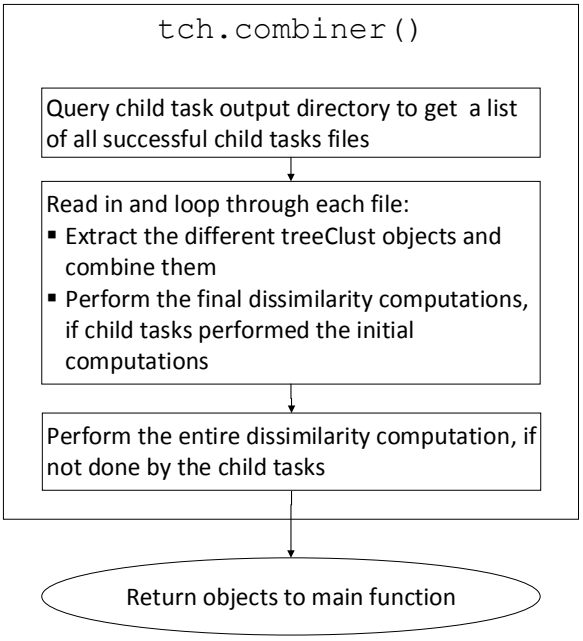


Figure 8. Computation Sequence: `tch.combiner()`

The combiner function's inputs include the directory where each child task output is saved, a `treeClust.control()` object, and information on how to perform the

dissimilarity computation. As Figure 8 shows, if the user elected not to perform the dissimilarity computation in parallel, it is done here. Once the steps in `tch.combiner()` are complete, we are ready to cluster and return our results to the user.

6. Parallelizing the Dissimilarity Functions

In order to make the computation of the dissimilarities occur in parallel we first modified the expressions for the underlying dissimilarities d_1, d_2, d_3 and d_4. We show our changes to d_1 and d_3 first, because they are simpler.

a. Tree Distance One and Three, d_1 and d_3

The changes to dissimilarity measure d_1—shown in equations (3.6) and (3.7)—is relatively simple. The changes to d_3 are the same as the changes to d_1. There are no changes to the tree-dissimilarities. However, we sum over the outputs of each tree, t, inside of each child task r, as in Equation (3.6), to give $\delta_1^r(i,j)$ the dissimilarity between pairs of observations i and j aggregated over all trees computed in child task r. The set of trees created by a single task is represented by X_r. We then sum over the outputs of each child task, r, with the parent task, Equation (3.7). R is the number of child tasks executed by `tch.childTaskRunner()` that result in output (it is possible for a task with very few trees to have every tree be discarded, leaving the task with no output). Only d_1 is indicated in equations (3.6) and (3.7), but the computations for d_3 are the same.

$$\delta_1^r(i,j) = \frac{\sum_{t \in X_r} d_1^t(i,j)}{X_r} \tag{3.6}$$

$$d_1^R(i,j) = \frac{\sum_{r=1}^{R} \delta_1^r(i,j)}{R} \tag{3.7}$$

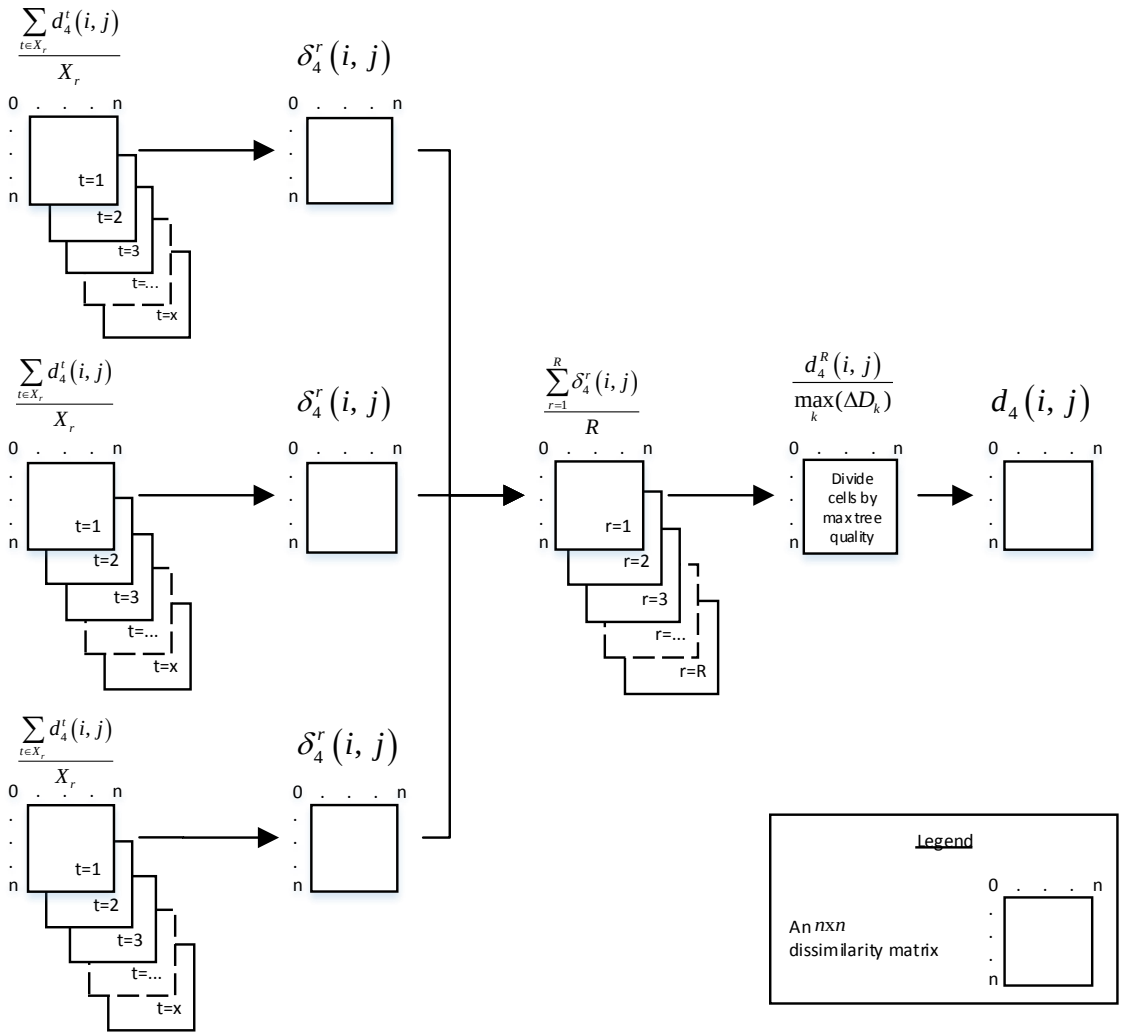
a. Tree Distance Two and Four, d_2 and d_4

The change to dissimilarity measure d_2 – shown in equations (3.8), (3.9), and (3.10) – requires changing when we scale by the maximum tree quality. The changes to d_4 are similar to the changes to d_2. Equations (3.8) and (3.7) show that we have removed scaling by the maximum tree quality during the dissimilarity computations performed by a child task, and have moved it to equation (3.10). Like equation (3.5), equation (3.8) shows the summation of the child tasks dissimilarity computations by the parent task. This is done because the child tasks' lack access to every tree, and therefore lack access to the maximum tree quality number. Despite this, we can still perform a piece of the dissimilarity computation. Figure 9 is included as a visual representation of how the overall process works, using `tcdist()` and d_4 as an example.

$$d_2^t(i,\ j) = \begin{cases} 0 & \text{if } L_t(i) = L_t(j) \\ \Delta D_t & \text{if } L_t(i) \neq L_t(j) \end{cases} \tag{3.8}$$

$$\delta_2^r(i,\ j) = \frac{\sum_{t \in X_r} d_2^t(i,\ j)}{X_r} \tag{3.9}$$

$$d_2(i,\ j) = \frac{\sum_{r=1}^{R} \delta_2^r(i,\ j)}{\mathrm{R} \times \max_k(\Delta D_k)} \tag{3.10}$$



Each task is assigned x trees. The task computes the d_4 distance for each tree, resulting in a single $n \times n$ lower triangular matrix of distances for each tree. These matrices for each tree are summed, and the final distance matrix for the task is saved. The parent task then reads in and sums all child task matrices, resulting in a final d_4 distance matrix, which is scaled using the maximum tree quality. Using separate tasks allows for the simultaneous computation of large matrices quickly, saving substantial time. Computation of all matrices sequentially can take several hours for big data sets, so breaking up this computation can drastically reduce run time.

Figure 9. Computation Sequence: `tcdist()` in Parallel Using d_4

a. Code changes

The changes required to make the two pairs of dissimilarity functions run in parallel are nearly the same. For each function, we add an additional logical argument where a true value indicates that dissimilarity is being computed in parallel. For distance metrics d_1 and d_3 this has no effect on computations that occur inside the dissimilarity functions. For d_2 and d_4 this results in the weights changing from including maximum tree quality to only including the current tree quality. The changes to `tcnewdata()` code are shown in Figure 10. To see the full function code, and the code changes for `tcdist()`, see Appendix A.

```
#-----------------d1 & d2-----------------------------------
#
# For d1, use daisy() on the "mat" object; for d2, do the same
# but with weights.
#
if (d.num == 1 || d.num == 2) {
    if (!missing (obj) && any (names (obj) == "mat"))
        mat <- obj$mat
    else
        if (missing (mat))
            stop ("For d1 or d2, this function requires the 'mat' element")

    for (i in 1:length(leaf.counts)) {
        mod <- model.matrix(~factor(mat[, i]) -  1)
        if (d.num == 2)
        {
          #UPDATE OCCURS HERE
          if(child.proc)
            newdata[, start[i]:end[i]] <- mod * tbl[i, "DevRat"]
          else
            newdata[, start[i]:end[i]] <- mod * tbl[i, "DevRat"]/ max(tbl[, "DevRat"])
        }else newdata[, start[i]:end[i]] <- mod #assign the model
    }
    return (newdata)
}
.
.
.
#-----------------d3 & d4-----------------------------------
#
for (i in 1:length (trees)) {
    leaf.dists <- d3.dist (trees[[i]], return.pd=TRUE)
    newcols <- cmdscale (leaf.dists, ncol(leaf.dists) - 1)

    #
    # The row.names of "newcols" are the leaf numbers, not the numbers
    # found in the "where" element of the tree. So we convert...
    #
    w <- trees[[i]]$where
    r <- row.names (trees[[i]]$frame)[w]

    #
    # ...and then extract the correct rows of newcols.
    #
    if (d.num == 4)
      #UPDATE OCCURS HERE
      if(child.proc)
        newdata[, start[i]:end[i]] <- newcols[r,] * tbl[i, "DevRat"]
      else
        newdata[, start[i]:end[i]] <- newcols[r,] * tbl[i, "DevRat"]/max(tbl[, "DevRat"])
    else
      newdata[, start[i]:end[i]] <- newcols[r,]
}
```

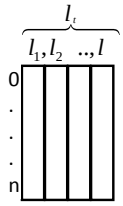
Changes are highlighted in blue; all other code is in black, and comments are in grey.

Figure 10. Function `tcnewdata()` Code Changes

For the `tch.combiner()` function, computing a dissimilarity function in parallel changes the number of elements that must be combined for each new output file it reads. Distance dissimilarity objects are always an $n(n-1)/2$ vector so they are simply added together until every file has been read, after which the final dissimilarity object is scaled, if necessary, using the maximum tree quality, $\max_k(\Delta D_k)$.

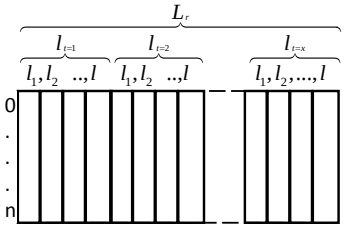
When `newdata` is computed, each tree contributes a set of l_t leaves, which become l_t columns in the `newdata` object (Buttrey & Whitaker, 2015). Therefore, every task will create a `newdata` object with L_r columns. To combine the `newdata` output from all tasks we use the `cbind()` function to get a final `newdata` object with the proper dimensions, $n \times L$, where L is the number of unique leaves across all trees in the data set. This final dissimilarity object is scaled, if necessary, using the maximum tree quality in the same manner as `tcdist()`. Figure 11 is included as a visual representation of this process.

In order to build the newdata object each tree with l leaves, contributes a set of l columns

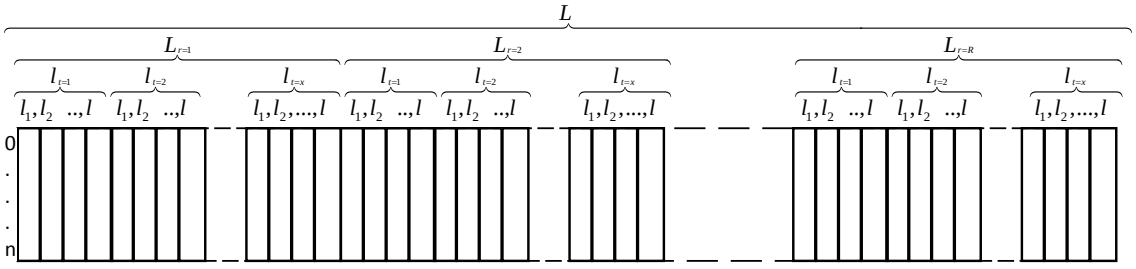


$$l \subseteq l_t \ \forall \ t$$

For a particular task, *r*, this results in a newdata object with L_r columns



Then, for the set of tasks, *R*, each task's newdata object is bound together using `cbind()`, resulting in the final $n \text{ x } L$ newdata object



Every entry in any of those columns is either a zero or the tree quality (Buttrey & Whitaker, 2015). If scaling by the maximum tree quality is required, it is done after the entire $n \text{ x } L$ newdata matrix is assembled.

Figure 11. Computation Sequence: `tcnewdata()` in Parallel

C. USER EXECUTION SEQUENCE

Running `tch.treeClust()` on the HSC is not as straightforward as running `treeClust()` on a local machine. So we provide a detailed description of the steps required to run `tch.treeClust()` on the HSC in Figure 12.

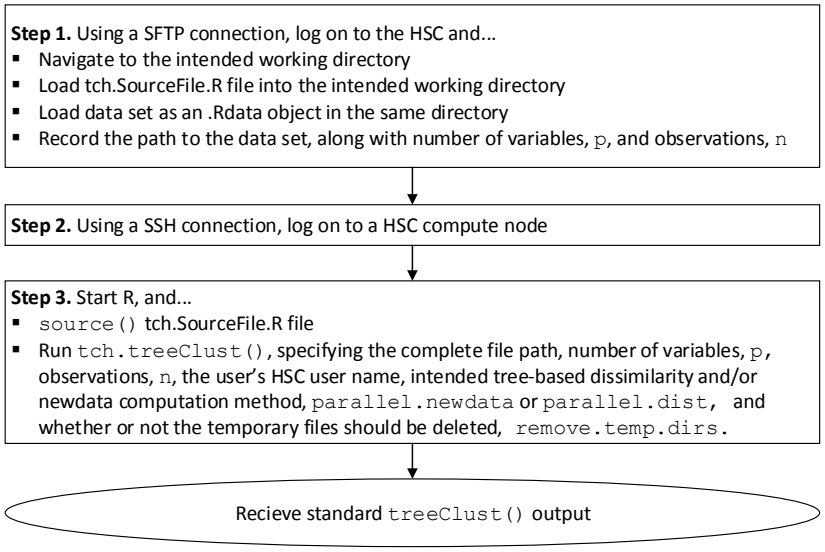


Figure 12. User's Execution Sequence

An example of Figure 12's step two and three is included in Figure 13.

```
[itmckech@submit-0 ~]

[itmckech@submit-0 ~]$ srun -n 10 –mem-per-cpu=2000 -t 01:00:00 --pty bash

[itmckech@compute-7-3 ~]$ cd work/itmckech/Thesis/experiments/internetAds/

[itmckech@compute-7-3 ~]$ R
.
.
.
>source('work/itmckech/Thesis/experiments/internetAds/tch.treeClustSource.R')

>path = 'work/itmckech/Thesis/experiments/internetAds/adsData.RData'

>inet_ads.tc = tch.treeClust(dfx = path, final.algorithm = 'kmeans', k = 2, p = 1558,
                             n = 3279, user.name = 'itmckech',
                             parallel.newdata = TRUE, remove.temp.dirs = FALSE)
.
.
```

Figure 13. User's Execution Sequence: Steps 2 and 3

THIS PAGE INTENTIONALLY LEFT BLANK

IV. DESIGN OF EXPERIMENTS

A. INTRODUCTION

We intend to determine five things through our analysis. First, whether or not our method effectively reduces run time. Second, whether or not there is a tradeoff between partitioning the data and adding more cores or tasks. Third, the optimal number of tasks to use when processing a data set. Fourth, whether or not using parallel processing to create the dissimilarity object provides a meaningful reduction in run time. Finally, fifth, if our memory constraint from out INLP formulation accurately predicts memory overflow. We will also establish values for our upper and lower bound heuristics based the results of our analysis.

To answer these questions, we designed a test consisting of 1260 experiments, using two data sets acquired from the University of California Irvine (UCI) Machine Learning Repository (Lichman, 2013a; Lichman, 2013b). Details about the data sets are given next.

B. DATA SETS

1. Internet Ads Data

This data set was acquired from the UCI Machine Learning Repository (Lichman, 2013b; Kushmerick, 1998), and contains $p = 1558$ variables and $n = 3279$ observations. It was previously used by Buttrey and Whitaker (2015) when testing the existing parallel processing method of `treeClust()`, providing us with a baseline of eight minutes on a local machine running 16 cores. Additionally, Buttrey and Whitaker provide a serial run time baseline of 57 minutes on a local machine with this data set.

2. Human Activity Recognition Using Smartphones Data

This data set was also acquired from the UCI Machine Learning Repository (Lichman, 2013a; Anguita, Ghio, Oneto, Parra, & Reyes-Ortiz, 2013), and contains $p = 561$ variables and $n = 10299$ observations. This data set is referred to as the HAR data from now on.

C. ASSUMPTIONS AND CONSTRAINTS

In order to run our tests we make a few assumptions and set a few constraints.

(1) Assumption One: Observation Size in Bytes

For both of our data sets the individual observations are numbers, and most numbers in R are of type double, which has a size of eight bytes (Grolemund, 2014). Therefore, we assumed an observation size of eight bytes, i.e. $BpO = 8$.

(2) Assumption Two: Core Clock Speed

We assume the cores that make up the HSC to be homogenous in processing capability.

(3) Constraint One: Memory per Core

We choose 2000 MB of memory per core (i.e. $MpC = 2000$) to ensure that we would experience memory overflow if we did not use enough tasks. This allowed us to test our memory overflow constraint, Equation (3.2).

(4) Constraint Two: Tasks per Core

For our tests we focused on using one task per core. We discuss exploring the possible benefits of using multiprogramming in further experiments in our future work section.

D. MEMORY FAILURE TESTS

In order to test our memory constraint we first use Equation (3.2) to estimate the largest value *x* can have without causing memory overflow. Then using Table 7 and Table 8, which are contained in Appendix B, we find the number of tasks associated with that value of *x*. In order to conduct sensitivity analysis on our constraint, we also test three values greater than *x*, and three values less than *x*. These values are also chosen from Table 7 and Table 8. The details of this test for each data set follows.

1. Internet Ads Data

The calculated maximum number of variables per task, using Equation (3.2):

$$floor\left(\frac{2000\times1000^2}{1558\times3279\times8}\right)=48$$

Using Table 7—contained in Appendix B—we find the corresponding number of tasks to use, and fill in the sounding design points, shown in Table 1. We replicate each design point ten times to determine if there is any variation in the outcome. This results in 70 experiments per data set.

Table 1. Memory Overflow Design Point Summary One: Internet Ads Data

Design Point	Number of Tasks	Number of Variables per Task, *x*	Expected Result
1	36	44	Success
2	35	45	Success
3	34	46	Success
4	33*	48	Success
5	32	49	Memory Overflow
6	31	51	Memory Overflow
7	30	52	Memory Overflow

*Design point four is the predicted smallest number of tasks we can use for the Internet Ads data set.

2. HAR Data

The calculated maximum number of variables per task, using Equation (3.2):

$$floor\left(\frac{2000\times1000^2}{561\times10299\times8}\right)=43$$

Using Table 8—contained in Appendix B—we find the corresponding number of tasks to use, and fill in the sounding design points, shown in Table 2. We replicate each design point ten times to determine if there is any variation in the outcome. This results in 70 experiments per data set.

Table 2. Memory Overflow Design Point Summary Two: HAR Data

Design Point	Number of Tasks	Number of Variables per Task, *x*	Expected Result
1	17	33	Success
2	16	36	Success
3	15	38	Success
4	14*	41	Success
5	13	44	Memory Overflow
6	12	47	Memory Overflow
7	11	51	Memory Overflow

*Design point four is the predicted smallest number of tasks we can use for the HAR data set.

E. RUN TIME TESTS

In order to get an idea of how many tasks we should use, our design points are staggered at approximate percentages of *p*, which are shown in Table 3 for the Internet Ads data, and Table 4 for the HAR data. We use approximate percentages of *p*, because there are only so many ways *p* can be evenly divided. We refer to these divisions as feasible partitions. For the Internet Ads data set, there are 78 different feasible partitions, and for the HAR data there are 47 feasible partitions. Tables including every feasible partition for both data sets are included in Appendix B for each data set; we also test the first feasible partition that does not result in memory overflow. For every design point, we test computing only the trees in parallel, and then we test computing the tress and the dissimilarity object in parallel. Finally, because the run time for any program is random, we run each design point 30 times in order to get an idea of how the run times are distributed.

Table 3. Run Time Design Point Summary One: Internet Ads Data

Design Point	~% of *p*	Number of Tasks	Number of Variables per Task, *x*
1	100%	1558	1
2	50%	779	2
3	25%	390	4
4	20%	312	5
5	15%	260	6
6	10%	156	10
7	7.5%	120	13
8	5%	82	19
9	2.5%	40	39
10	2.11%*	33	48

*33 tasks is the calculated smallest number of tasks we can use without experiencing memory overflow with the Internet Ads data.

Table 4. Run Time Design Point Summary Two: HAR Data

Design Point	~% of *p*	Number of Tasks	Number of Variables per Task, *x*
1	100%	561	1
2	50%	281	2
3	25%	141	4
4	20%	113	5
5	15%	94	6
6	10%	57	10
7	7.5%	44	13
8	5%	30	19
9	2.5%*	14	41

*14 tasks is the calculated smallest number of tasks we can use without experiencing memory overflow with the HAR data.

THIS PAGE INTENTIONALLY LEFT BLANK

V. ANALYSIS

A. INTRODUCTION

This chapter presents the results of our new parallel processing method, placing particular emphasis on comparing the run times of only computing the trees in parallel vs computing the trees and the dissimilarities in parallel. It is important to note that in both approaches we compute the trees and the dissimilarities. When we only compute the trees in parallel, the dissimilarities are computed sequentially after all of the trees have been computed.

We start with our memory overflow results, and then move on to the run time analysis. In addition to total run time, the run times for the child tasks are evaluated, and the time the combiner function takes to read and aggregate the child tasks output is evaluated.

B. MEMORY OVERFLOW RESULTS

For both data sets, use of the estimated value of x (x=48 for the Internet Ads data set, and x=41 for the HAR data set) resulted in memory overflow. This is shown in Table 5 and Table 6. Our sensitivity analysis indicates that the number of tasks used needs to be strictly greater than the number of tasks associated with the estimated value of x given by Equation (3.2). For the Internet Ads data set this means that we have to use 34 or more tasks ($x \leq 46$) in order to avoid memory overflow. For the HAR data set this means we have to use 15 or more tasks ($x \leq 38$) in order to avoid memory overflow. Based on the results of these tests, we modified the last design point of each data set's run time experiments.

Table 5. Memory Overflow Results: Internet Ads Data

Design Point	Number of Tasks	Number of Variables per Task, *x*	Expected	Result
1	36	44	Success	Success
2	35	45	Success	Success
3	34	46	Success	Success
4	33	48	Success	Memory Overflow
5	32	49	Memory Overflow	Memory Overflow
6	31	51	Memory Overflow	Memory Overflow
7	30	52	Memory Overflow	Memory Overflow

Table 6. Memory Overflow Results: HAR Data

Design Point	Number of Tasks	Number of Variables per Task, *x*	Expected	Result
1	17	33	Success	Success
2	16	36	Success	Success
3	15	38	Success	Success
4	14	41	Success	Memory Overflow
5	13	44	Memory Overflow	Memory Overflow
6	12	47	Memory Overflow	Memory Overflow
7	11	51	Memory Overflow	Memory Overflow

C. RUN TIME RESULTS

For each run time experiment, we first examine the results of the Internet Ads data set, and then the HAR data set.

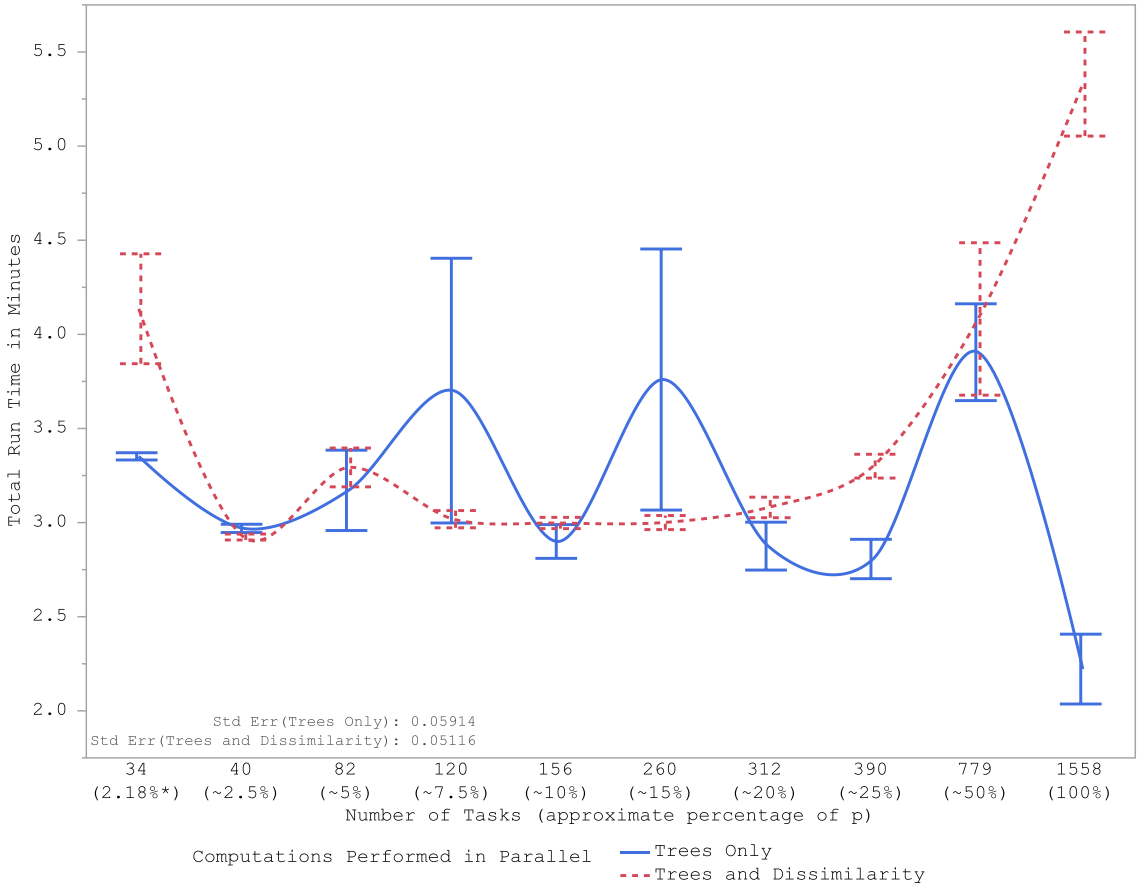
1. Total Run Time

In order to best understand our results we first studied the mean total run time of each design point, and the 95% confidence interval (CI) for each design point. The 95% CI for the mean total run time of each experiment are not consistent (the CIs are depicted as error bars in Figures 14, 17, 20, 22, 24, 25, 26, and 27), so we then looked at the run time distribution for specific design points, focusing on those with abnormally wide confidence intervals. We then examined the median run time for each design point, to determine what run time we should expect for each design point.

Each data set's run time oscillates substantially as we increase the number of tasks, and generally, each design point has normally distributed run-times. This leads us to believe that the oscillation in run time realized across all design points occurs for one of two reasons. First, it has to do with the tradeoff of partitioning the data and using more cores that we mentioned in Chapter II. When our method uses more cores, it has to split the data into smaller chunks. These chunks then have to be distributed across the HPC, which takes substantial time. If each additional task (or core, because we using one core per task) does not substantially reduce the run time for each task, then the time required to access additional nodes or racks on the HSC becomes a burden on the total run time. This event is most visible when the number-of-tasks $\approx 50\% \times p$ for the Internet Ads data set, and when number-of-tasks $\approx 25\% \times p$ for the HAR data set. The other reason that we could be realizing run time oscillations is that 30 repetitions per design point is not enough, and the oscillations may be less apparent with more repetitions.

a. Internet Ads Data

The average total run time by task is shown in Figure 14. When computing only the trees in parallel, the execution time appears to oscillate substantially as the number of tasks increases, before reaching its lowest time at p tasks. When computing the trees and the dissimilarities in parallel, the execution exhibits a 'U' shaped curve as the number of tasks increases. The 95% CIs of the expected total run time at 120 and 260 tasks, when computing only the trees in parallel, seem unusually wide so we investigated the distributions of the total run times for these two design points. These are shown in Figure 15.



Each error bars is constructed using a 95% CI of the mean total run time for each design point. *Lower bound provided by our memory-based constraint.

Figure 14. Average Total Run Time: Internet Ads Data

In both cases where the CI is very wide, the data was not normally distributed, and was affected by a group of high run-time outliers. We believe that because the majority of the observations are much lower than the average, that this is likely the result of the HSC receiving higher user traffic for a short period, forcing it to distribute the computations to nodes on different racks. For this reason, we then looked at the median run time.

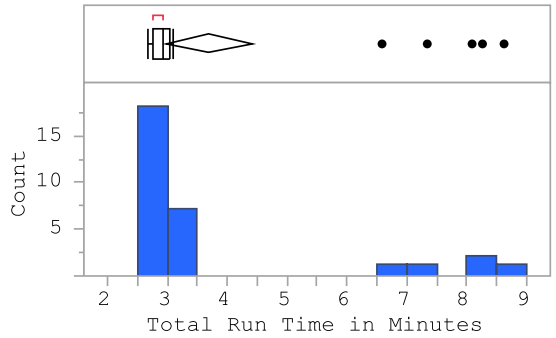
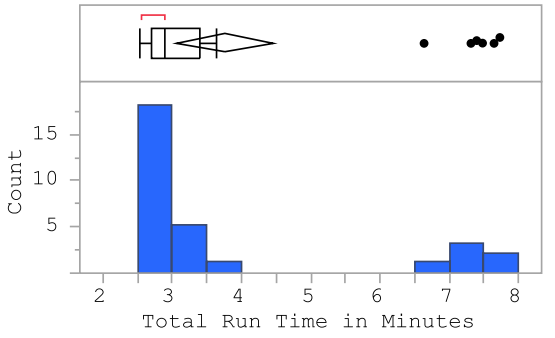
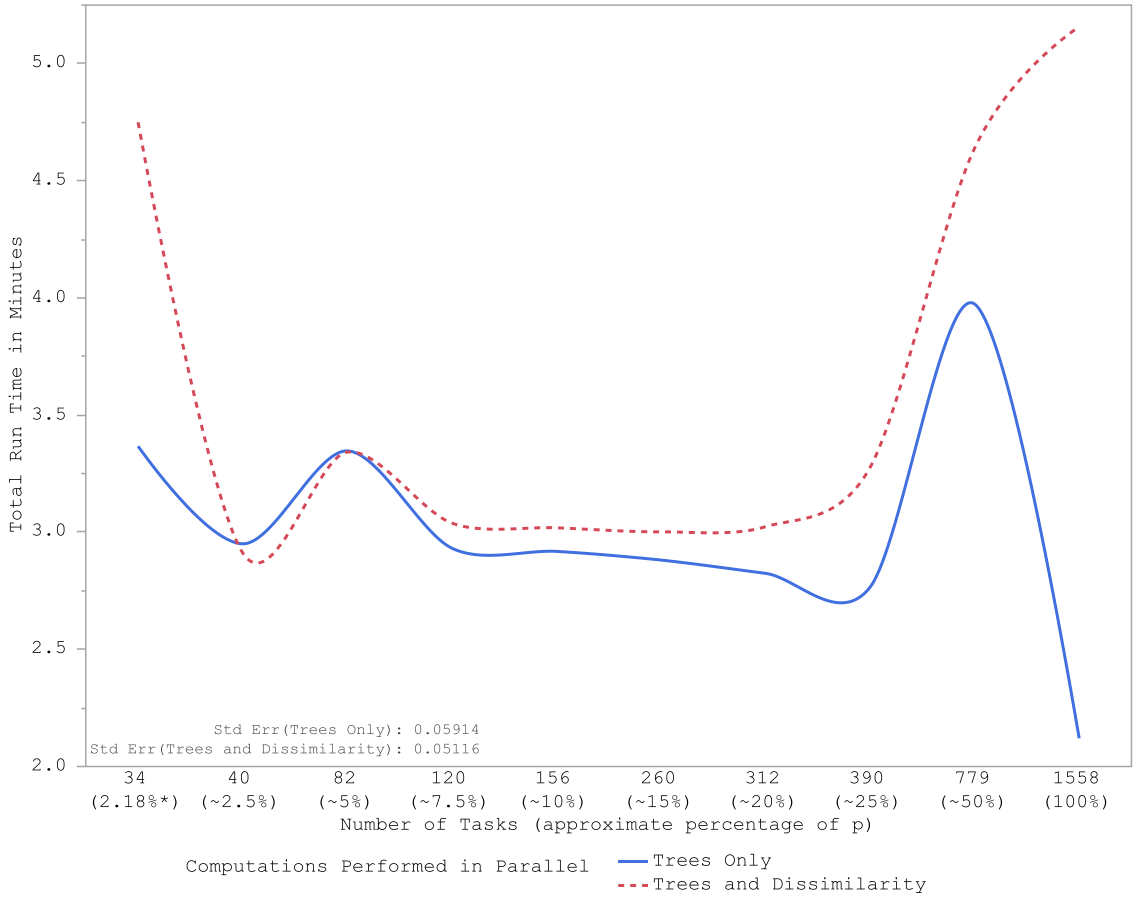
120 Tasks (~7.5% of p), Only Computing Trees in Parallel	260 Tasks (~15% of p), Only Computing Trees in Parallel
	

Figure 15. Select Run Time Distributions Internet Ads Data

The median run time in Figure 16 shows that no true oscillation in run time occurs at 120 and 260 tasks; however, oscillations do occur at 82 (~5% of p) and 390–779 (~25-50% of p) tasks. The design points at these oscillations are normally distributed around the mean. The median run time also shows that both methods follow a similar trend of generally decreasing run times as the number of tasks increase, until $\text{number-of-tasks} \approx 25\% \times p$. At ~25% both methods begin to increase rapidly in run time. When computing only the trees in parallel the method realizes its best median run time at p tasks. When computing the trees and the dissimilarities in parallel, the method realizes its worst median run time at p tasks. We will show in a following section that this is possibly due to over-partitioning the data.



*Lower bound provided by our memory-based constraint.

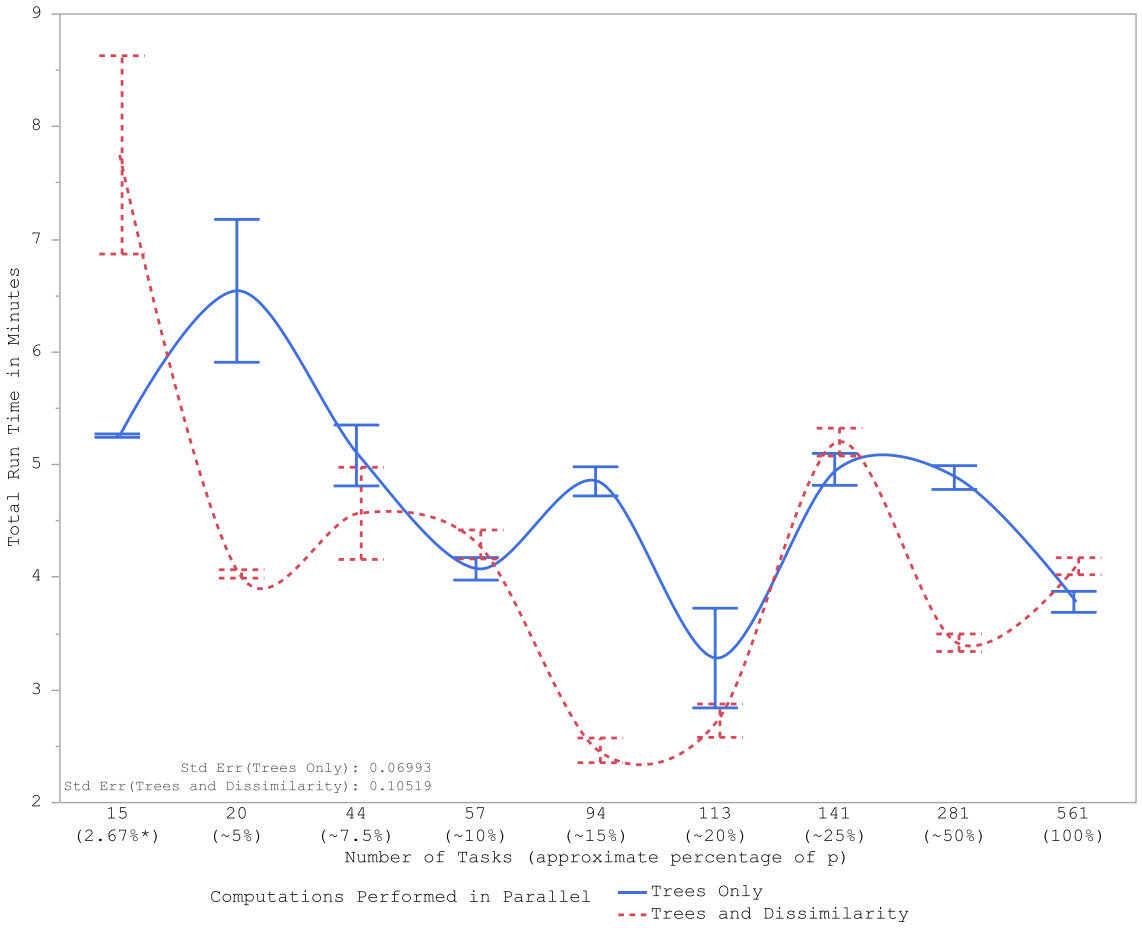
Figure 16. Median Total Run Time: Internet Ads Data

b. HAR Data

The average total run time is shown in Figure 17. Again when computing only the trees in parallel, the run time oscillates as the number of tasks approaches *p*. There is also an additional hump at ~15% when computing only the trees in parallel, and we are not sure why this additional hump occurs only for one of the two methods. This time a substantial drop off in run time occurs before the number of tasks reaches *p* when computing both the trees and the dissimilarity function in parallel. Both methods also exhibit a spike in run time at ~25%. Interestingly, both methods realize their best run time when the number of tasks used is less than *p*.

Once again, we believe the oscillation in run time has to do a tradeoff between partitioning the data, and using more cores. For this data set, using p tasks excessively distributes the data causing an increase in run time compared to using a fraction of p.

We believe that the fact that n is much greater than p for the HAR data set is likely the reason that computing both the trees and the dissimilarity function in parallel performs better here than it did for the Internet Ads data set.



Each error bars is constructed using a 95% CI of the mean total run time for each design point.*Lower bound provided by our memory-based constraint.

Figure 17. Average Total Run Time: HAR Data

We examined the run time distributions, and found several instances where the data is not normally distributed, and appears to form two run-time groups. This is shown in Figure 18. Again, we believe this is due to what nodes our tasks were allocated at run time, and node allocation is subject to other users being on the HSC.

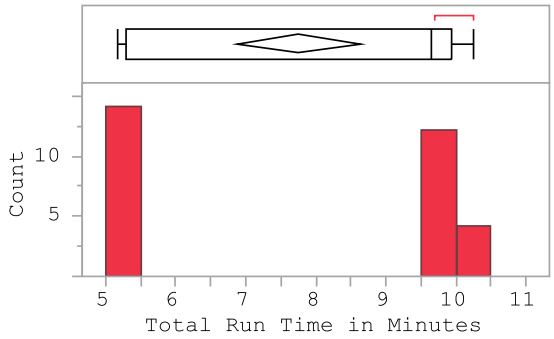
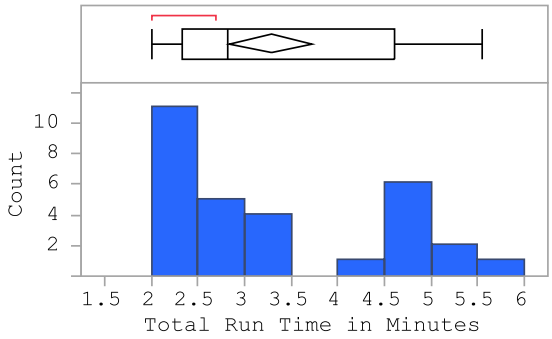
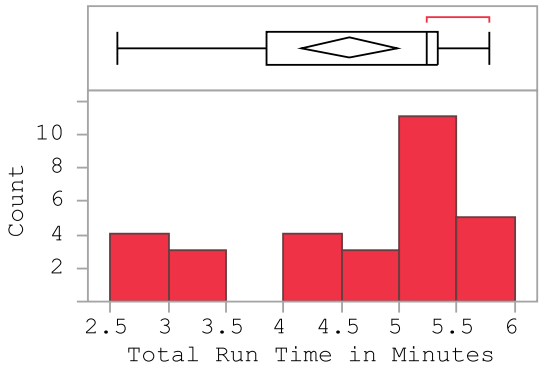
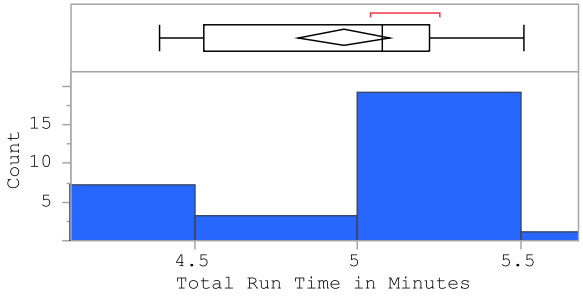
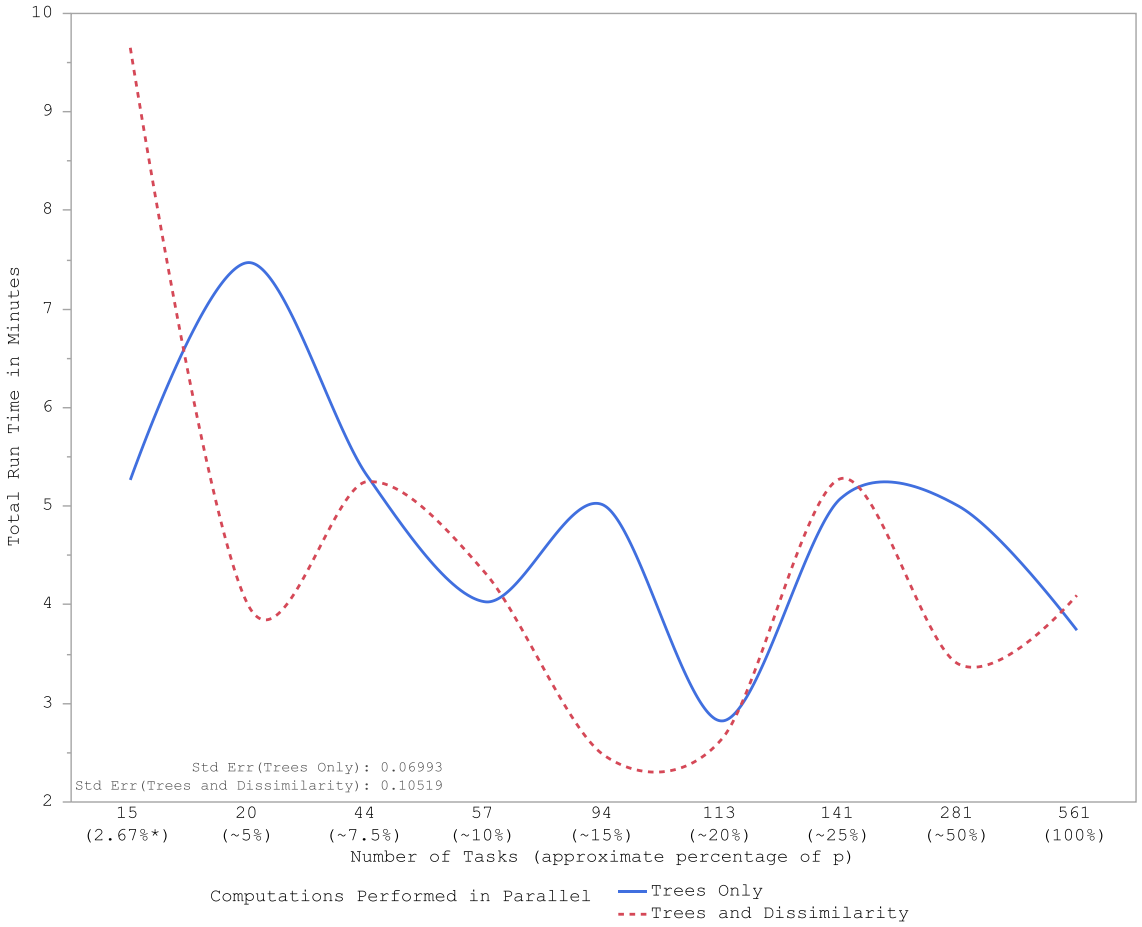
15 Tasks (2.67% of p), Computing Trees and the Dissimilarity Function in Parallel	113 Tasks (~20% of p), Only Computing Trees in Parallel
	
44 Tasks (~7.5% of p), Computing Trees and the Dissimilarity Function in Parallel	**141 Tasks (~25% of p), Only Computing Trees in Parallel**
	

Figure 18. Run Time Distributions: HAR Data

For the most part the run time CIs are narrower than they were for the Internet ads data. Because of this, we do not see as significant a change when we look at the median run time shown in Figure 19.



*Lower bound provided by our memory-based constraint.

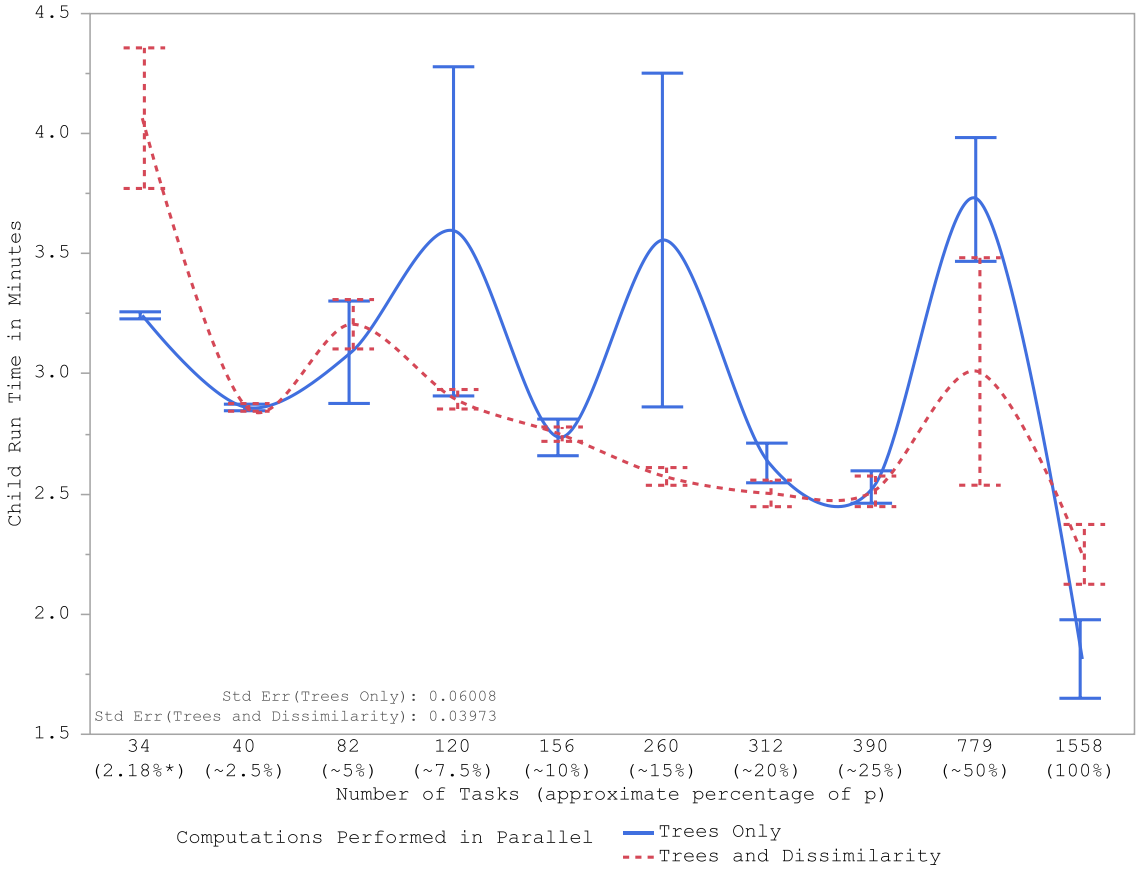
Figure 19. Median Run Time: HAR Data

2. Child Task Run Time

The child tasks run time exhibit the same oscillatory behavior seen in the total run time, so we again look at the mean and median run times for each design point.

a. Internet Ads Data

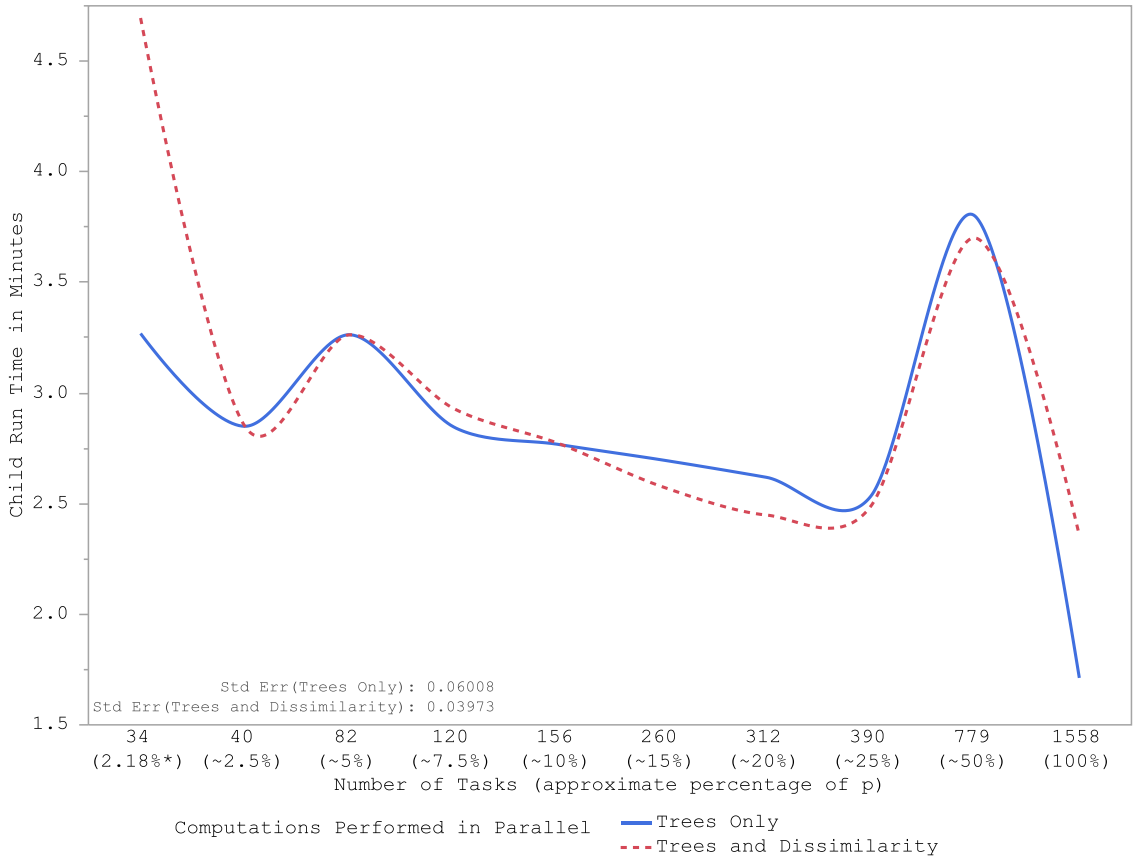
The average execution time of the child tasks, shown in Figure 20, indicates that the fluctuations in total run time are due to the child tasks. Figure 20 also shows that the spike in run time realized when computing the trees and the dissimilarity function in parallel is not due to the child tasks.



Each error bars is constructed using a 95% CI of the mean total run time for each design point. *Lower bound provided by our memory-based constraint.

Figure 20. Average Child Task Run Time: Internet Ads Data

Looking at the median run time (Figure 21) shows that the two methods follow the same trend. Both have a decreasing trend, and experience spikes in run time at around 5% of *p*, and then again at 25–50% of *p*. They also both experience a significant dip at *p* tasks.

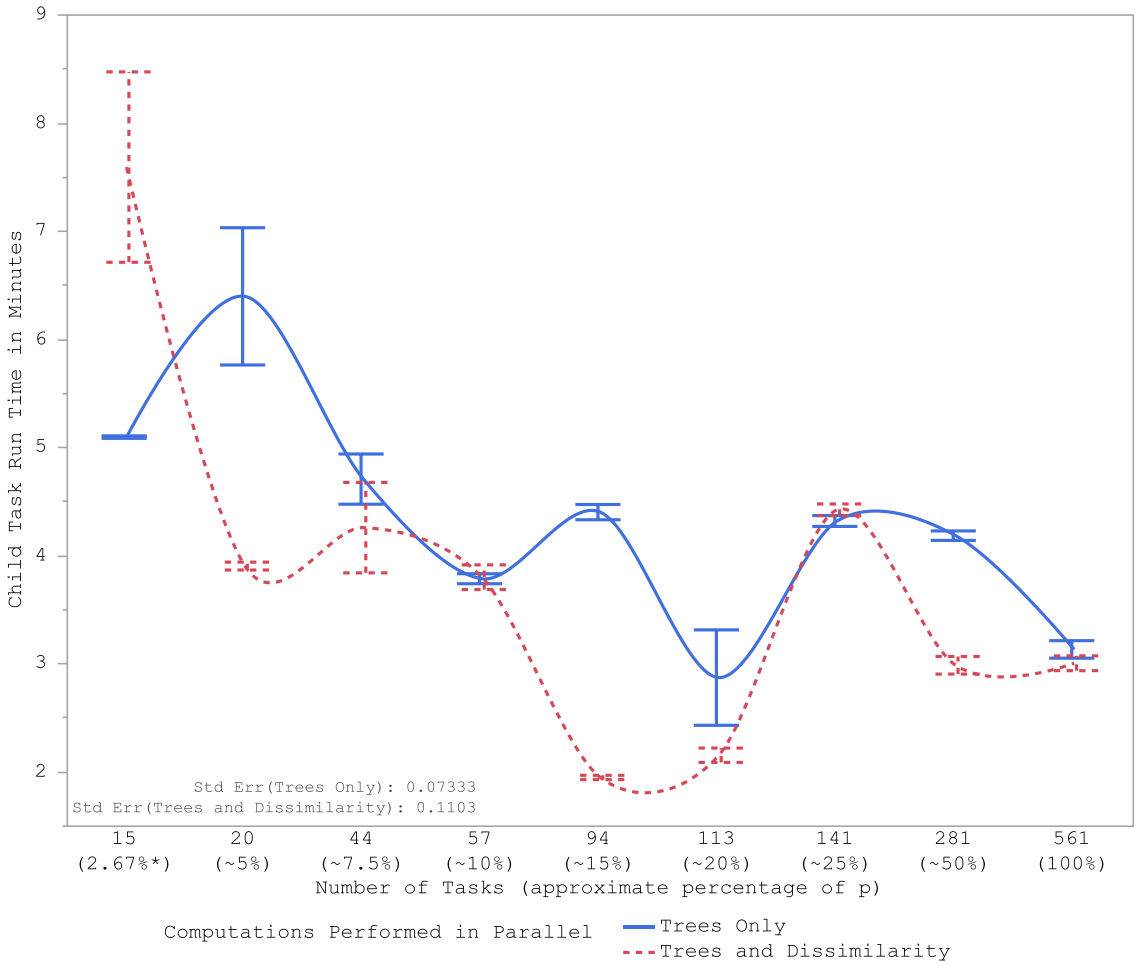


*Lower bound provided by our memory-based constraint.

Figure 21. Child Tasks Median Run Time: Internet Ads Data

b. HAR Data

The average execution time of the child tasks, shown in Figure 22, looks nearly identical to the average total run time shown in Figure 17.

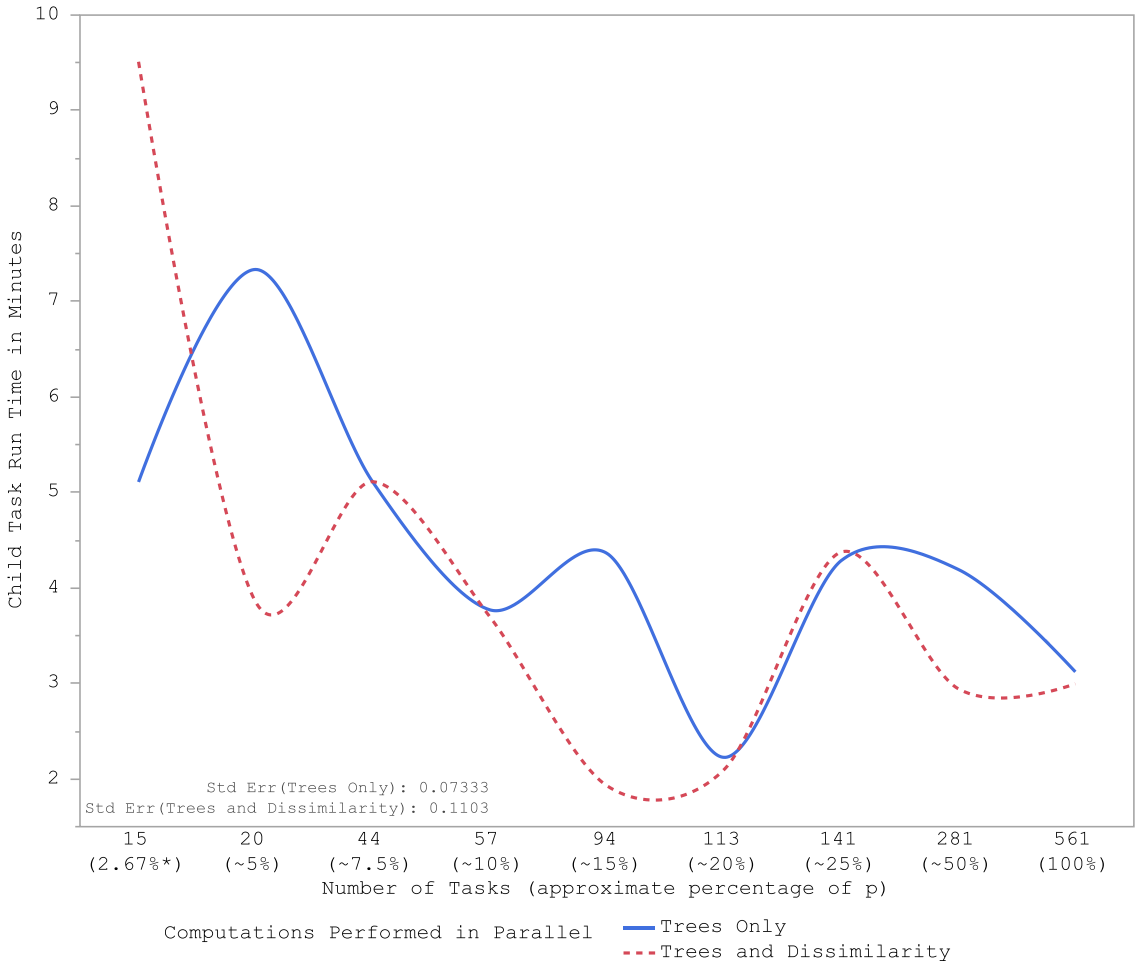


Each error bars is constructed using a 95% CI of the mean total run time for each design point. *Lower bound provided by our memory-based constraint.

Figure 22. Average Child Task Run Time: HAR Data

Aside from the design point at 2.67% of *p*, computing the trees and dissimilarity function in parallel appears to consistently provide the best run time for the HAR data set. Both data sets also indicate a downward sloping trend as the number of tasks increases, and both sets also experience a spike in run time at, or shortly before, we get to 50% of *p*

tasks. Computing only the trees in parallel also results in more oscillatory behavior than computing both the tress and the dissimilarity function.



*Lower bound provided by our memory-based constraint.

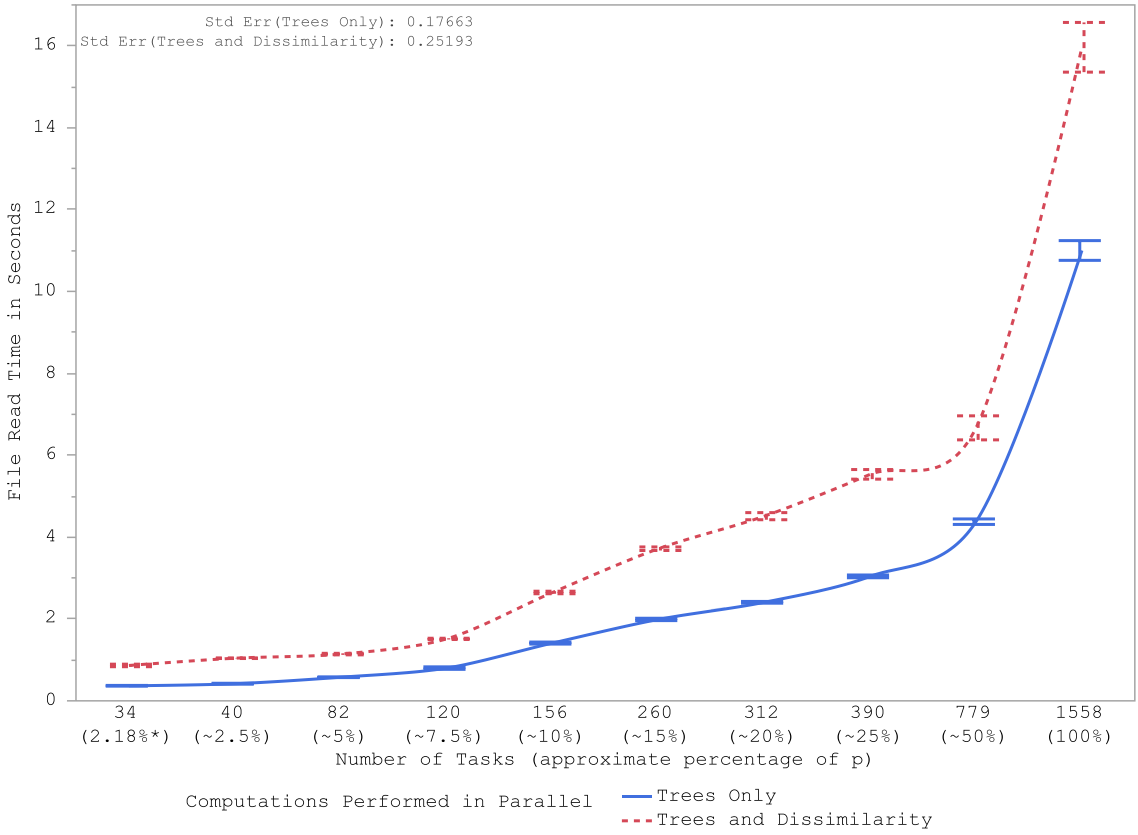
Figure 23. Median Child Task Run Time: HAR Data

3. Combiner Function Time

The combiner functions run time follows a generally increasing trend, so we only examined the mean run time for each design point. For each data set, we looked at the time spent reading in files, and the time spent combing R objects from those files.

a. File Read Time

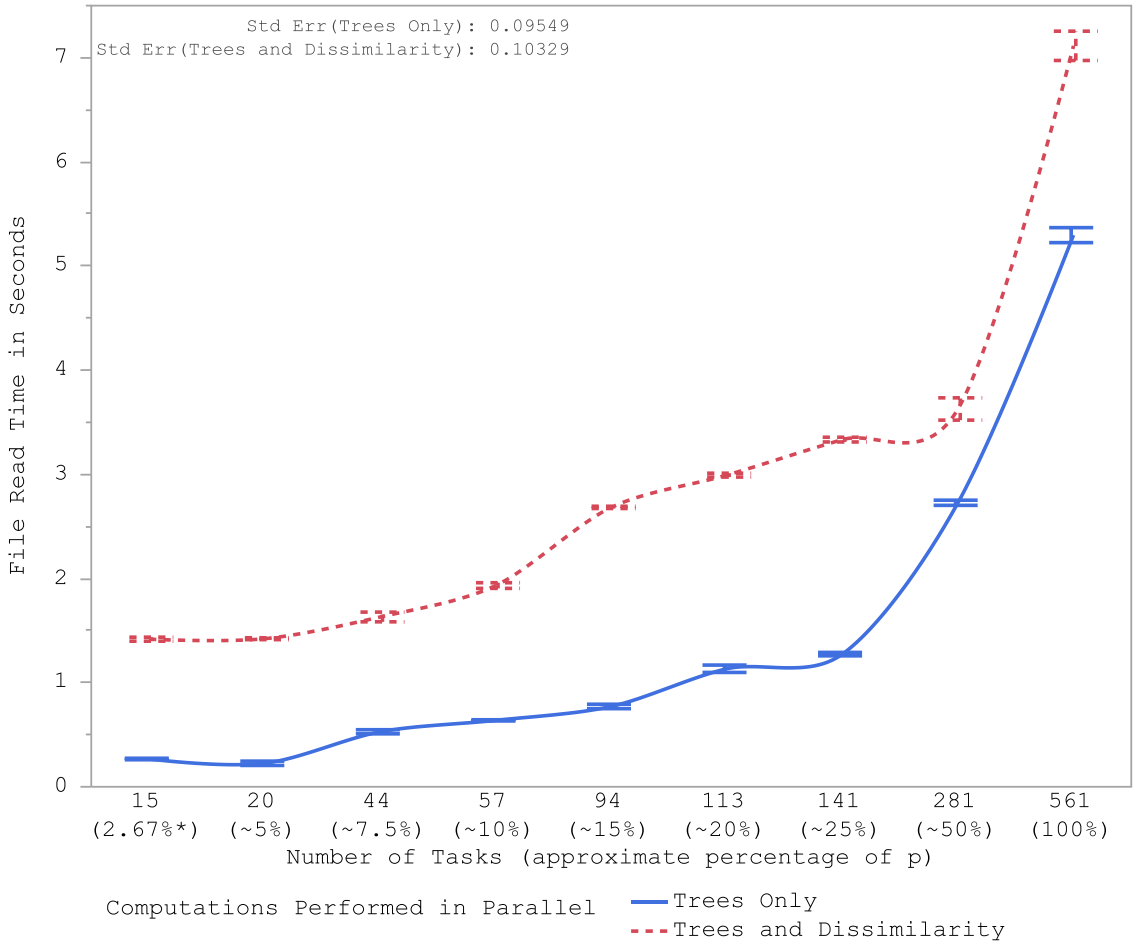
Since read and write operations require reading from non-volatile memory, which is very slow, we looked at the total amount of time each design point spent reading from memory. Average read times for both data sets are included in Figure 24 (Internet Ads), and Figure 25 (HAR).



Each error bars is constructed using a 95% CI of the mean total run time for each design point. *Lower bound provided by our memory-based constraint.

Figure 24. Average File Read Time: Internet Ads Data

For both data sets, the read time increases as the number tasks increases, which was expected. The total amount of time-spent reading was generally a small amount of the total execution time for either method. As we would expect, reading in the data when we computed both the trees and the dissimilarity function in parallel took longer, because the files were larger.

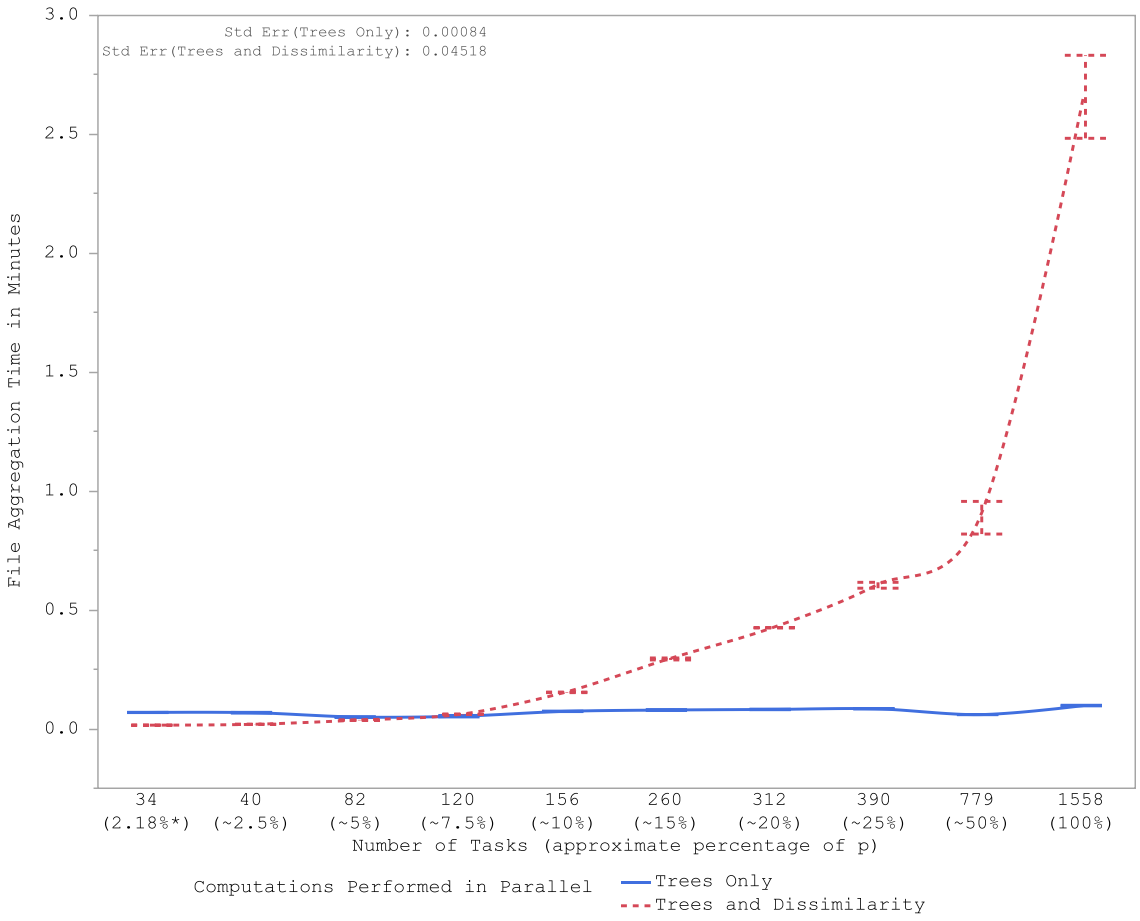


Each error bars is constructed using a 95% CI of the mean total run time for each design point. *Lower bound provided by our memory-based constraint.

Figure 25. Average File Read Time: HAR Data

b. File Aggregation Time

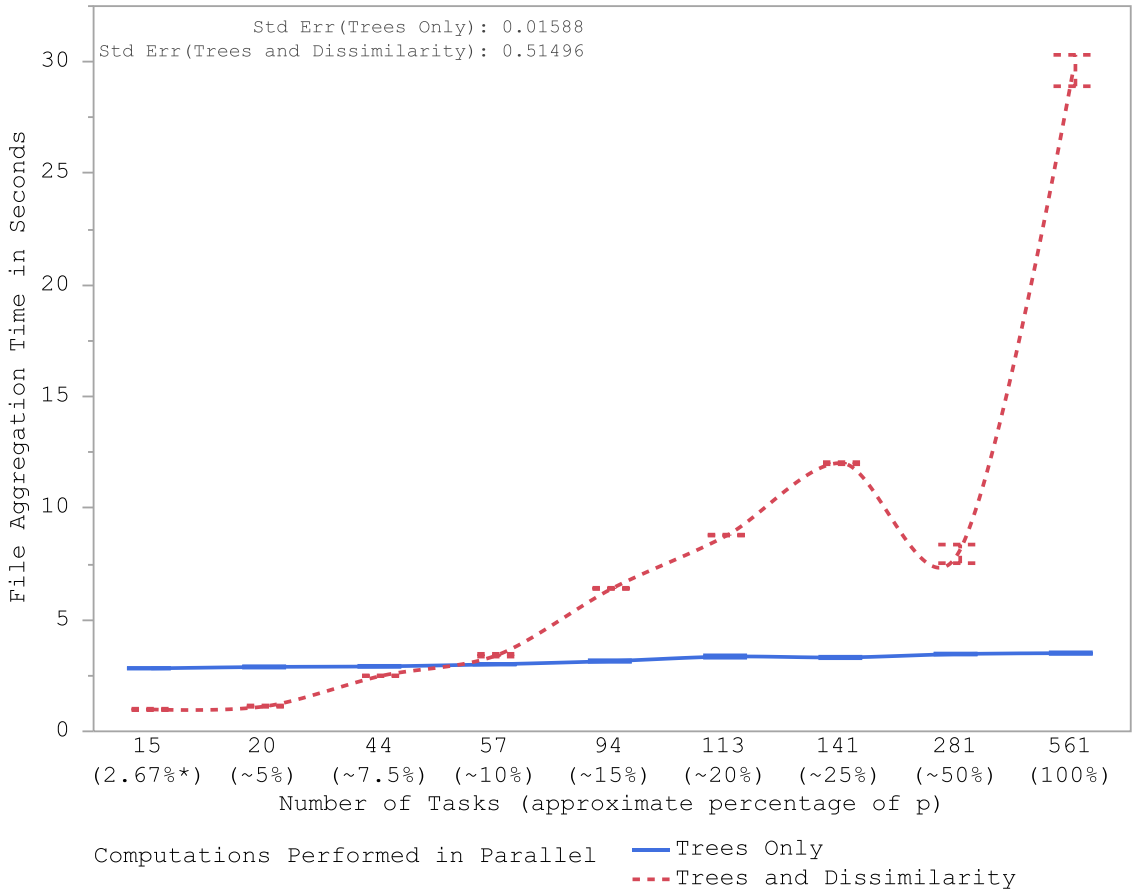
As we mentioned previously, computing both the trees and the dissimilarity function in parallel required an increasing, and substantial, amount of time to combine each file, as the number of tasks increased. This is shown in Figure 26 (Internet Ads), and Figure 27 (HAR). This is because the dissimilarity objects are large, and when the number of tasks is equal to *p*, this process takes up the majority of the tasks' run time. When computing only the trees in parallel, the files are relatively small so the time required to combine them does not increase significantly.



Each error bars is constructed using a 95% CI of the mean total run time for each design point. *Lower bound provided by our memory-based constraint.

Figure 26. Average File Aggregation Time: Internet Ads Data

For the HAR data, the total amount of time spent aggregating files was a considerably smaller portion of the total run time than it was for the Internet Ads data. We believe this is due to the different ratios of $\frac{p}{n}$ for the two data sets. Because the HAR data has many more observations than factors, it benefits it to compute the dissimilarity function in parallel. Both data sets also saw their aggregation time nearly triple when p tasks were used, and the dissimilarity function was computed in parallel.



Each error bars is constructed using a 95% CI of the mean total run time for each design point. *Lower bound provided by our memory-based constraint.

Figure 27. Average File Aggregation Time: HAR Data

D. CONCLUSIONS

For our conclusions, we address the five questions we set out to answer with our analysis, we derive values for our two heuristics, Rub and Rlb, and we update our INLP formulation to reflect our results.

(1) Method Effectiveness

Our method appears to provide an effective way to perform parallel processing with treeClust on the HSC. For the Internet Ads data set, all experiments also realized a faster run time than using treeClust's existing parallel processing method on a personal computer.

(2) Possible Effects of Data Partitioning

For both data sets, the total run time oscillates as the number of tasks (and cores) increases. However, the run time realized a generally decreasing trend as the number of tasks used increased, until $\text{number-of-tasks} \approx 25\% \times p$. We believe the oscillation in run time possibly has to do with a tradeoff between partitioning the data, and using more cores. When our method uses more cores, it has to split the data into smaller chunks. These chunks then have to be distributed across the HPC, which takes substantial time. If each additional task does not substantially reduce the run time for each task, then the time required to access additional nodes or racks on the HSC becomes a burden on the total run time.

(3) Determining the Optimal Number of Tasks

For both data sets only computing the trees in parallel resulted in good run time for p tasks. However, the run time for p tasks was not consistently the fastest. When the $\text{number-of-tasks} \approx 20\% \times p$ for the HAR data set, the median run time was faster than the run time for $\text{number-of-tasks} = 100\% \times p$.

When computing both the trees and the dissimilarity function in parallel the Internet Ads data set realized a drastic increase in run time when p tasks were used. For the HAR data set the fastest median run time occurred when $\text{number-of-tasks} \approx 15\% \times p$.

Since p tasks only resulted in the fastest median run time in a quarter of our experiments we believe the optimal number of tasks is a fraction of p. This realization results in changes to both of our heuristics, Rub and Rlb, which follow.

(4) Setting the Default Heuristics Values

For both data sets there was a spike in run time realized when the number of tasks was approximately 5–7.5% of p. For this reason, we assign the following default value to our lower bound heuristic:

$$Rlb = 0.075$$

For both data sets there was also a spike in run time realized when the number of tasks was approximately 25–50% of p. For this reason, we assign the following default value to our upper bound heuristic:

$$Rub = 0.20$$

(5) Computing the Dissimilarity Function

Using parallel processing to compute the dissimilarity function has mixed results. Based on the data sets used, it appears that as n grows processing the dissimilarity function in parallel becomes more beneficial. When n is small `treeClust()` is better off only computing the trees in parallel.

(6) Evaluating the Memory Constraint

Our memory based constraint needs to be adjusted in order to precisely determine when memory overflow will occur. The estimated number of trees any particular task could build resulted in memory overflow for both data sets. This means that we need to use more tasks. Based on the results of our experiments, Equation (5.1) uses the memory estimate from Equation (3.5) to estimate the specific number of tasks we must use to avoid memory overflow. It specifies that the number of tasks must be strictly greater than the calculated number possible.

$$R > ceiling\left(\frac{p}{floor\left(\frac{MpC \times 1000^2}{p \times n \times BpO}\right)}\right) \tag{5.1}$$

VI. SUMMARY AND FUTURE WORK

A. SUMMARY

The purpose of this thesis was to provide a parallel processing method for treeClust that works on the HSC. Chapter I introduced the concepts involved with this thesis. Chapter II described in detail the background information related to our problem. Chapter III detailed our approach, and changes from what was previously in place. Chapter IV described the data sets, and methods we used to test our method. Chapter V presented the results, and conclusions of our experiments.

In order to make treeClust run on the HSC we made use of large-scale batch scripting and a parent/child task architecture. To further reduce run time, we modified the tree-based dissimilarity measures and treeClust's dissimilarity functions in order to compute them in parallel. We then performed extensive testing using two data sets in order to develop a heuristic based INLP that determines the optimal number of tasks to use for a specific data set.

The vast majority of our 1260 experiments show that our method outperformed the existing non-HPC parallel processing method. This indicates that our specialized approach provides an extensible set of functions that allows future users to take full advantage of the HSC with treeClust. Based on our experiments we were able to refine our INLP to choose the optimal number of tasks to use when processing a large data set.

B. FUTURE WORK

1. Additional Testing

The first step in future work is testing the method on additional data sets to further refine our INLPs heuristics. The optimal number of tasks is likely related to the ratio $\frac{p}{n}$. In both of the data sets we used $p < n$, but this is not always the case. It is likely that the optimal number of tasks changes when $p > n$ or $p = n$. It is also likely that different behavior will be realized when $p << n$ and $p >> n$.

The optimal number of tasks is also likely related to the type of data (numeric, categorical, or mixed), and to how the dissimilarity function is computed (in serial or in parallel).

2. Selecting Specific Nodes on the Hamming Supercomputer

The two run-time groups realized for several design points indicate that the set of nodes used for treeClust's computations plays a very large role in the overall run time of our method. A mitigation technique, to ensure that data partitions are not spread out as far, would be to select specific HSC nodes and ensure each node is located on the same rack. This minimizes the physical distance the data must travel, ensuring a significantly faster run time. The SLURM command `sinfo` allows us to do this.

`sinfo` supplies a breakdown of the entire HSC and how it is currently being used. Regular expressions can then be used to get the racks and nodes not in use, and then select the best subset. This subset of nodes can then be passed to SLURM along with our child task resource request using the `nodeslist` argument of the `sbatch` command.

3. Other High Performance Computers

The second step in future work is extending our method to other HPCs. Our method is dependent on the HSC's job scheduling software, SLURM, and the Bash command language. Any HPC using the same software architecture should be able to make use of our method, but this will need to be verified. Our method could also be extended to HPC-like resources, such as a cloud computing service. Amazon Web Services (AWS) is one such example.

AWS provides a cheap and easily accessible HPC-like resource for military researchers not located at NPS. Specifically, our method could be easily modified to run on AWS Batch, which is the cloud computing batch job service hosted by AWS. Like SLURM, AWS Batch allows a program, or user, to specify resource requirements (CPUs and memory), and it can handle hundreds or thousands of jobs, making a good corollary to the HSC (Amazon Web Services, Inc., 2017).

4. GPUs

GPUs may provide a nice alternative the CPUs because of how many cores are available without having to access more nodes or racks. This could alleviate the non-normality realized when other users are active on the HSC. Additionally, multithreading is enabled on the HSC's GPU cores (Haferman, 2017), providing another possible avenue for a speed increase.

5. Multiprogramming

Multiprogramming allows for better utilization of scarce CPU resources, and can decrease total run time when partitioning data adds excessive overhead computation time. Two situations may occur that could make incorporating multiprogramming into our approach a good idea. First, the HSC is a shared resource and we will not have exclusive access to all of its computational power. Therefore, we need to make the most efficient use of the resources we do have, and multiprogramming allows us to do that. Second, many big data sets will have many more variables than we have cores; multiprogramming could allow us to process more variables in a shorter amount of time, and the SLURM allows users to specify exactly how many tasks they want to run on each core at a time (SLURM Workload Manager, 2017).

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. FUNCTION CODE

This appendix contains the R code for each new and modified treeClust function. For each function, regular comments are in grey, comments about updates and modifications are in blue, and code is in black.

(1) `tch.treeClust()` Function Code

```
###############################################################################
#
# Name:
#  tch.treeClust()
#
# Description:
#  This function is a modification of treeClust()
#  This function serves as a main - or parent - function. It is run in the
#  parent task and oversees the instantiation and execution of child tasks.
#  Child tasks are used to build trees and compute dissimilarity. It should be
#  considered a "shell" of treeClust, because most of treeClust()'s functional
#  code has been moved to other functions. Clustering is still done in this
#  function.
#
# Change Highlights:
#  Changes are indicated by ###UPDATED HERE###
#  When it is practical the change is included in comments directly above
#  the affected line.
#
# List of Modifications:
#  -error checking of many inputs is performed by child tasks
#
#  -trees are built by tch.childTaskRunner()
#
#  -dist & newdata are computed by tch.childTaskRunner() and tch.combiner()
#
# New Arguments:
# -p  : the number of variables (columns) in the data set,
#     default: no default, user input required
#
# -n  : the number of observations (rows) in the data set,
#     default: no default, user input required
#
# -user.name: the user's hamming account username, must be exact with no spaces
#     default: no default, user input required
#
# -parallel.newdata: logical, compute newdata in parallel
#     default: FALSE
#
# -parallel.dists: logical, compute tree-based dissimilarity in parallel
#     default: FALSE
#
# -remove.temp.dirs: logical indicating if the temporary directories created by
#   the function should be deleted. These directories can be useful for error
#   checking.
#      default: TRUE
#
# Return type class is the same as treeClust():
#  treeClust
#
###############################################################################
tch.treeClust <- function(dfx,
                          d.num = 1,
```

```
                          final.algorithm,
                          k,
                          control = tch.treeClust.control(),
                          rcontrol = rpart.control (),
                          verbose = FALSE,
                          p,
                          n,
                          user.name,
                          parallel.dists = FALSE,
                          parallel.newdata = FALSE,
                          remove.temp.dirs = TRUE,...)
{

if(verbose == TRUE)
  cat('FUNCTION: tch.treeClust \n')

###UPDATED HERE###
#  Changed check for data frame, to a check for
#   a path with a specific data frame included
if(!is.character(dfx))
    stop("This function requires the full path to an .RData object containing \n
          a data frame - name of data frame with \."RData\" included. \n
          DO NOT ATTEMPT TO PASS IN A DATA FRAME.")

#
# We need to extract the name of the object we're looking
# for, and where it's located. We will then move into that
# directory.
#
dfx.name = gsub("*/|.RData.*","","path) #get the name of .RData object
re = paste0(dfx.name,'.*')
dfx.path = gsub(re,","path)#get the path
setwd(dfx.path)

###UPDATED HERE###
# Removed check for names with embedded spaces,
#  because we don't have the data frame yet

#
# Final.algorithm has to be agnes, pam, clara, or kmeans (or missing).
#  For everything except agnes, a k is required.
#
if (!missing (final.algorithm)) {
    if (!is.element (final.algorithm, c("agnes," "pam," "clara," "kmeans")))
        stop ("Unrecognized final algorithm")
#
# For pam, clara or k-means, "k" must be present. For agnes, set it to -1 if
# it's missing.
#
    if (missing (k))
        if (is.element (final.algorithm, c("kmeans," "pam," "clara")))
            stop ("With kmeans, pam or clara, specify the number of clusters 'k'")
        else k <- -1
        if (k == -1 & control$cluster.only == T) stop ("Cluster.only TRUE requires k")

}#end if

#
# determine the number of processes to use  (R), and the number of trees to
# build per process (x)
#
run.settings = tch.determineSettings(p, n, control, verbose = verbose)

# Create three new directories to store the output from all child tasks.
new.dirs = tch.createTempDirs(dfx.name, run.settings[1], verbose = verbose)

#
# we are in a new working directory!!
#
```

```
if(verbose == TRUE)
  cat('FUNCTION: tch.treeClust \n')

#
# Run child processes
#
tch.childTaskRunner(dfx.name = dfx.name,
                        d.num = d.num,
                        final.algorithm = final.algorithm,
                        k = k,
                        control = control,
                        user.name = user.name,
                        R = run.settings[1],
                        x = run.settings[2],
                        p = p,
                        parallel.newdata = parallel.newdata,
                        parallel.dists = parallel.dists,
                        new.dirs = new.dirs,
                        verbose = verbose)

if(verbose == TRUE)
  cat('FUNCTION: tch.treeClust \n')
#
# Combine output from child tasks
#
combiner = tch.combiner(d.num = d.num,
                         final.algorithm = final.algorithm,
                         control = control,
                         parallel.newdata = parallel.newdata,
                         parallel.dists = parallel.dists,
                         child.dir = new.dirs$child.dir,
                         verbose = verbose)

if(verbose == TRUE)
  cat('FUNCTION: tch.treeClust \n')

#
# Now, if there's a final algorithm, call it.
#
if (missing (final.algorithm)) {
    final.algorithm <- "None"
    final.clust <- NULL
} else {
#
# We call "agnes" or "pam" by "do.call," which saves a copy of the dists
# in the call element. That thing is huge and unnecessary, so we remove it.
#
    if (final.algorithm == "agnes") {
        final.clust <- do.call (final.algorithm, list (x = combiner$dists, ...))
        final.clust$call$x <- "deleted"
        if (control$cluster.only == TRUE)
            final.clust <- cutree (final.clust, k = k)
    }#end if
    if (final.algorithm == "pam") {
        final.clust <- do.call (final.algorithm, list (x = combiner$dists, k = k, ...))
        final.clust$call$x <- "deleted"
        if (control$cluster.only == TRUE)
            final.clust <- final.clust$clustering
    }#end if
#
# k-means are clara use the "newdata" data, which has p-1 columns for each
# tree with p leaves.
#
    if (is.element (final.algorithm, c("clara," "kmeans"))) {
        if (final.algorithm == "kmeans")
            final.clust <- kmeans (x = combiner$newdata, centers = k)
        else
            final.clust <- clara (x = combiner$newdata, k = k)
```

```
        if (control$cluster.only == TRUE)
                final.clust <- final.clust$cluster
    }#end if
} # end of "final algorithm" stuff


#
# If we were only asked for the clustering, return that.
#
if (control$cluster.only)
    return (final.clust)
#
# Set up return value; add requested stuff.
#
return.val <- list(call = match.call(), d.num = d.num, tbl = combiner$tbl,
                   extended.tbl = combiner$extended.tbl,
                   final.algorithm = final.algorithm, final.clust = final.clust)

if (control$return.trees)
    return.val$trees <- combiner$big.list.of.trees
if (control$return.dists)
    return.val$dists <- combiner$dists
if (control$return.newdata)
    return.val$newdata <- combiner$newdata
if (control$return.mat)
    if (!missing (final.algorithm) &&
        is.element (final.algorithm, c("clara," "kmeans")))
        return.val$mat <- combiner$newdata
    else
        return.val$mat <- combiner$mat
class(return.val) <- "treeClust"

setwd(new.dirs$org.dir)

if(remove.temp.dirs == TRUE){
  delete.dir.cmd = paste0('rm -r ', new.dirs$run.dir)
  system(delete.dir.cmd)
}#end if

return(return.val)

}#end function
```

(2) `tch.determineSettings()` Function Code

```
################################################################################
#
# Name:
# tch.determineSettings()
#
# Description:
# This function uses an iterative search to approximate the optimal number of
# tasks to start for a given data set.
#
# Arguments:
# -p  : the number of variables (columns) in the data set
#     default: no default, user input required
#
# -n  : the number of observations (rows) in the data set
#     default: no default, user input required
#
# -control: a tch.treeClust.control() object
#
# -verbose: This argument is used for error checking, it forces function to
#   print to the screen so the user knows that the function call occurred
#
# Arguments from tch.treeClust.control():
#
# -MpC: memory per core
#     default: 3700. The median amount of RAM available per core on the hamming
#              supercomputer in megabytes.
#
# -BpO: bytes per observation
#     default: 8. We initially assume the data is made up of numbers, and
#              numbers in R typically are of class numeric, which has a size of
#              8 bytes.
#
# -Rub: Number of Tasks, upper bound. Given as a fraction of p
#     default: 0.20, this value is a heuristic based upper bound established
#              through testing. Multiplied by p, it gives the upper bound on the
#              number of tasks to start.
#
# -Rlb: Number of Tasks, lower bound. Given as a fraction of p
#     default: 0.075, this value is a heuristic based lower bound established
#              through testing. Multiplied by p, it gives the lower bound on the
#              number of tasks to start.
#
# Return:
# [1] R, the number of child tasks to start
# [2] x, the number of trees created by each task
#
################################################################################
tch.determineSettings = function(p, n, control = tch.treeClust.control(), verbose =
FALSE)
{

if(verbose == TRUE)
  cat('FUNCTION: tch.determineSettings \n')

#constraint 2, Memory-based lower bound
maximum.x = floor((control$MpC*(1000^2))/(p*n*control$BpO))
mem.constraint = ceiling (p/maximum.x)

#heuristic upper and lower bounds
R.upper.bound = ceiling(control$Rub*p)
R.lower.bound = ceiling(control$Rlb*p)

#
# Find the optimal number of tasks to use through an
# iterative search
#
```

```
for(x in 1:maximum.x){

  # constraint 1
  R = ceiling(p/x)

  # constraint 3, starting variable of the
  #               second-to-last task
        second.to.last.task.start.col = x*(R-1)

        # check constraints 2, 3, 4, 5
        if(R > mem.constraint && second.to.last.task.start.col < p &&
     R <= R.upper.bound && R > R.lower.bound)
  {
    R.to.use = R
    x.to.use = x
    break
        }#end if
}#end loop

#Return optimal results to user
return.val = c(R.to.use, x.to.use)
class(return.val) = ('tch.determineSettings')
return (return.val)
}#end function
```

(3) `tch.childTaskRunner()` Function Code

```
################################################################################
#
# Name:
# tch.childTaskRunner()
#
# Description:
# This function writes a batch R file used to run separate tasks of R, which
# build trees using a specified range of variable from the data set. The SLURM
# environment variable SLURM_ARRAY_TASK_ID is used to iterate over the batch file.
#
# Arguments:
# -dfx.name: the name of the data frame to use
#
# -final.algorithm: which algorithm to use, supplied by the user to tch.treeClust()
#
# -k: the number of clusters to use, supplied by the user to tch.treeClust()
#
# -control: the treeClust control object, supplied by the user to tch.treeClust()
#
# -user.name: the name of the user, supplied by the user to tch.treeClust()
#
# -R: the number of child tasks to start, computed in tch.determineSettings()
#
# -x: the number of trees built per task, computed in tch.determineSettings()
#
# -p: the number of variables (columns) in the data set
#
# -parallel.dists: logical indicating if the child tasks should compute the
#    tree-distance
#
# -parallel.newdata: logical indicating fi the child tasks should compute
#    newdata
#
# -new.dirs: list of directories created by tch.createTempDirs()
#
# -verbose: This argument is used for error checking, it forces function to
#   print to the screen so the user knows that the function call occurred
#
# Return:
#  Void
#
#  Output is saved by the child tasks to new.dirs$log.dir and new.dirs$child.dir
#
################################################################################
tch.childTaskRunner = function(dfx.name,
                               d.num,
                               final.algorithm,
                               k,
                               control = tch.treeClust.control(),
                               user.name,
                               R,
                               x,
                               p,
                               parallel.newdata,
                               parallel.dists,
                               new.dirs,
                               verbose = FALSE)
{

if(verbose == TRUE)
  cat('FUNCTION: tch.childTaskRunner \n')

#move into the run directory, so that the batch file saves there
setwd(new.dirs$run.dir)

#
```

```
# Set-up the bash environment variable SLURM_ARRAY_TASK_ID
#       This is used to index through the tasks
#
SATID.cmd = paste0('set SLURM_ARRAY_TASK_ID = ', R)
system(SATID.cmd)

#
# Prepare batch file inputs
#
# cluster.only is not set, because the child tasks will not cluster
#
tree.return = FALSE
if(control$return.trees){
  tree.return = TRUE
}else if((parallel.dists == FALSE && control$return.dists == TRUE) &&
         (d.num == 3 || d.num == 4))
{
  tree.return = TRUE
}else if((parallel.newdata == FALSE && control$return.newdata == TRUE) &&
         (d.num == 3 || d.num == 4))
{
  tree.return = TRUE
}#end if

#
# Write the batch file - childBatch.txt - from the given
# inputs to treeClust, the computed number of tasks to
# start obtained from tch.determineSettings(), and the
# number of columns in the data set.
#
childBatch = paste0(
  '#!/usr/bin/Rscript',                                                  ' \n',
  'library(tree)',                                                       ' \n',
  'library(treeClust)',                                                ' \n\n',

  'source(\'', new.dirs$org.dir,'/','tch.SourceFile.R\')',             ' \n\n',

  'num = as.numeric (Sys.getenv (\'SLURM_ARRAY_TASK_ID\'))',           ' \n\n',

  'attach(\'', new.dirs$org.dir, '/', dfx.name, '.RData\', pos=2)',     ' \n\n',
  'start.col = (num-1) * ', x,' + 1',                                    ' \n',
  'end.col = num * ', x,                                                 ' \n',
  'if (end.col >', p,') end.col =', p,                                 ' \n\n',

  'out = tch.child.treeClust(dfx = ', dfx.name,                         ', \n',
  '       d.num = ', d.num,                                             ', \n',
  '       col.range = start.col:end.col',                               ', \n',
  '       final.algorithm = \'', final.algorithm,'\'',                  ', \n',
  '       k = ', k,                                                     ', \n',
  '       control = tch.treeClust.control(return.trees = ', tree.return, ', \n',
  '                  serule = ',          control$serule,               ', \n',
  '                  DevRatThreshold = ', control$DevRatThreshold,')',  ', \n',
  '       parallel.dists = ',   parallel.dists,                         ', \n',
  '       parallel.newdata = ', parallel.newdata,                       ', \n',
  '        verbose = TRUE)',                                          ' \n\n',

  'save (out, file = paste0(\'',new.dirs$child.dir,
         'out\' , num,  \'.RData\'))',                                   ' \n')


#now write that file and run it
fileConn = file("childBatch.txt")  #create file connection
writeLines(childBatch, fileConn)   #write childBatch to the file
close(fileConn)                    #close connection

#move into the log directory, so that log files save there
setwd(new.dirs$log.dir)

#make batch file runnable
```

```
d2u.cmd = paste0('dos2unix ',new.dirs$run.dir,'childBatch.txt')
system(d2u.cmd)

#change permissions to executable
chmod.cmd = paste0('chmod +x ',new.dirs$run.dir,'childBatch.txt')
system(chmod.cmd)

#set up run command, and run the batch file. This instantiates all child tasks.
run.cmd = paste0('sbatch --mem-per-cpu=',control$MpC, ' --array=1-', R,' ',
                 new.dirs$run.dir, 'childBatch.txt')
system(run.cmd)

#
# Wait until all child tasks finish
#

#gets the number of running processes for the user
check.cmd = paste0('squeue -u ', user.name,' | wc -l')

#if the number of running jobs is <= 2, all tasks are complete
ready=FALSE
while(!ready){
        Sys.sleep(5)
        ready = as.integer(system(check.cmd, intern = TRUE)) <= 2
}#end loop

#
# Now that all the child processes are finished and we
# won't generate any more log files go back to the
# run directory
#
setwd(new.dirs$run.dir)

}#end function
```

(4) `tch.createTempDirs()` Function Code

```
################################################################################
#
# Name:
# tch.createTempDirs()
#
# Description:
# Create three new directories to store the output from tch.treeClust()
#
# 1) The first directory, starting with "tch_run," holds all files produced by
#     this run of tch.treeClust().
#
# 2) The second directory, starting with "tch_log," stores all log output and
#    error output files from the child task
#
# 2) The third directory, starting with "tch_child," stores the R objects the
#    child task creates.
#
#    The rest of each directory name is a combination of:
#    -The name of the data object
#    -The number of tasks used
#    -the current time, format YYYY-MM-DD_HH:MM:SS
#
# Arguments:
# -dfx.name: the name of the data frame the user supplied to tch.treeClust()
#
# -R: the number of child tasks used
#
# -verbose: This argument is used for error checking, it forces function to
#   print to the screen so the user knows that the function call occurred
#
# Return:
# -A list containing four objects that hold the paths to the original directory
#  and the three new temporary directories
#
################################################################################
tch.createTempDirs = function(dfx.name, R, verbose = FALSE)
{

if(verbose == TRUE)
  cat('FUNCTION: tch.createTempDirs \n')

#save the original directory, we'll need it for reading the data
org.dir = getwd()

# get current time, put in appropriate format
current.time = Sys.time()
current.time = gsub(" ," "_," current.time)

# run directory
run.dir = paste('tch_run', dfx.name, R, current.time, sep="_")
run.dir = paste0(getwd(),'/',run.dir,'/')
dir.create(run.dir)
setwd(run.dir) #move into the run directory

# log directory
log.dir = paste('tch_log', dfx.name, R, current.time, sep="_")
log.dir = paste0(getwd(),'/',log.dir,'/')
dir.create(log.dir)

# child directory
child.dir = paste('tch_child', dfx.name, R, current.time, sep="_")
child.dir = paste0(getwd(),'/',child.dir,'/')
dir.create(child.dir)

# Build return list
new.dirs = list(org.dir = org.dir,
```

```
                run.dir = run.dir,
                log.dir = log.dir,
                child.dir = child.dir)

class(new.dirs) = ('tchDirectoryList')
return(new.dirs)
}#end function
```

(5) `tch.childTreeClust()` Function Code

```
################################################################################
#
# Name:
# tch.child.treeClust()
#
# Description:
#  This function is a modification of treeClust().
#  This function is intended to be called by tch.childProcessRunner(). It runs
#  the treeClust() function on a sub-set of the columns in the data set. Many of
#  the objects have been modified to account for this. This makes them smaller,
#  which is necessary for big data sets.
#
# Change Highlights:
#  Changes are indicated by ###UPDATED HERE###
#  When it is practical the change is included in comments directly above the
#  affected line.
#
# List of Modifications:
#  -many references to dfx, have been changed to
#   dfx[col.range], or col.range
#
#  -parallel processing using sockets has been removed
#
#  -clustering has been removed
#
#  -the return values have been modified to what is required
#   by tch.combiner().
#
# Return type class is now:
#  tch.child.treeClust
#
#  This is important because these returns will be saved as .”RData” objects
#  which the user may interact with. In that case we need to clearly identify
#  where they came from.
#
################################################################################
###UPDATED HERE### removed col.range default to change it to a required argument
#                  changed control to tch.treeClust.control()
tch.child.treeClust = function(dfx,
                               d.num = 1,
                               col.range,
                               final.algorithm,
                               k,
                               control = tch.treeClust.control(),
                               rcontrol = rpart.control (),
                               parallel.dists,
                               parallel.newdata,
                               verbose = FALSE, ...)
{

if(verbose == TRUE)
  cat('FUNCTION: tch.child.treeClust \n')

if(!is.data.frame(dfx))
    stop("This function requires a data frame")

###UPDATED HERE### dfx -> col.range
leaf.matrix <- as.data.frame(matrix(0., nrow(dfx), length(col.range)))
#
# Fail if any columns have names with embedded spaces.
#
###UPDATED HERE### dfx -> dfx[col.range]
dimnames(leaf.matrix) <- dimnames(dfx[col.range])
###UPDATED HERE### dfx -> dfx[col.range]
nm <- names(dfx[col.range])
if (length (grep (" ," nm) > 0))
```

```
    stop ("Some columns have embedded spaces in their names, and that's not good.")
#
# Final.algorithm has to be agnes, pam, clara, or kmeans (or missing).
# For everything except agnes, a k is required.
#
additional.args <- list (...)
if (!missing (final.algorithm)) {
    if (!is.element (final.algorithm, c("agnes," "pam," "clara," "kmeans")))
        stop ("Unrecognized final algorithm")
#
# For pam, clara or k-means, "k" must be present. For agnes, set it to -1 if
# it's missing.
#
    if (missing (k))
        if (is.element (final.algorithm, c("kmeans," "pam," "clara")))
            stop ("With kmeans, pam or clara, specify the number of clusters 'k'")
        else k <- -1
        if (k == -1 & control$cluster.only == T) stop ("Cluster.only TRUE requires k")

}
#
# Set up the two-column "result" matrix.
#
###UPDATED HERE### dfx -> dfx[col.range]
results <- matrix(0., nrow = ncol(dfx[col.range]), ncol = 2.)
###UPDATED HERE### dfx -> dfx[col.range]
dimnames(results) <- list(dimnames(dfx[col.range])[[2.]], c("DevRat," "Size"))
#
# Set up the big list of trees if asked -- except that we will need
# that list no matter what, when d.num == 4 or we're asked for "newdata,"
# which includes the cases where we plan to cluster with "clara" or "kmeans."
# We also need it for return.dists = TRUE and d.num = 3. Save the fact of
# us keeping the list of trees to "control" for later use.
#
control$keep.trees <- FALSE
if (control$return.trees || control$return.newdata || d.num == 4 ||
    (!missing (final.algorithm) &&
     is.element (final.algorithm, c("clara," "pam," "kmeans"))) ||
    (control$return.dists && d.num == 3)) {
    control$keep.trees <- TRUE
    ###UPDATED HERE### dfx -> dfx[col.range]
    big.list.of.trees <- vector("list," ncol(dfx[col.range]))
}

#
# Loop over columns. For each column, call treeClust.rpart() to build
# a tree. Fill the "results" matrix with the DevRat and Size entries.
# Keep the trees if (a) asked or (b) we want to compute "newdata" or
# (c) we want to compute d4.
#
###UPDATED HERE### Removed the parproc option completely.
###UPDATED HERE### i in col.range -> i in 1:length(col.range)
for(i in 1:length(col.range)) {
  if(verbose > 0)
    cat("Creating rpart tree with column," i, "\n")
  out.i <- treeClust.rpart(i, dfx, d.num, control, rcontrol)
  results[i,] <- c(out.i$DevRat, out.i$Size)
  if (results[i, "Size"] > 1) {
    leaf.matrix[,i] <- out.i$leaf.where
    if (control$keep.trees || (d.num == 3 || d.num == 4))
      big.list.of.trees[[i]] <- out.i$tree
  }
}

###UPDATED HERE###
#   (control$keep.trees) -> (control$keep.trees || (d.num == 3 || d.num == 4))
#
if (control$keep.trees)
  big.list.of.trees <- big.list.of.trees[results[,"Size"] > 1]
```

```
leaf.matrix <- leaf.matrix[,results[,"Size"] > 1, drop = F]

if(!any(results[, "Size"] > 1))
    stop("No tree produced anything! Panic!")
#
# Save original results. For the moment, we will keep both the full set
# of results and the one that shows only trees that were kept. This
# is a little redundant, but we want (# row in result) to be equal to
# (# of trees).
#
original.results <- results
results <- results[results[,"Size"] > 1,, drop=F]
#
# If there's a final algorithm, prepare for it now. Agnes and pam use
# the inter-point distances, plus any additional arguments, so we'll
# compute those. Also compute them if we have asked for them explicitly.
#

#
### UPDATED HERE### changed conditional to parallel.dists.
#                   Original command not repeated because it's too long.
# Compute dists if needed. d1 and d2 don't use the trees.
if (parallel.dists)
    ### UPDATED HERE, now using tch.tcdist. Added argument child.task
    dists <- tch.tcdist (tbl = results, mat = leaf.matrix,
                         trees = big.list.of.trees, d.num = d.num,
                         child.task = parallel.dists)

#
### UPDATED HERE### changed conditional to parallel.newdata.
#                   Original command not repeated because it's too long.
# K-means and clara use "newdata." Compute that thing if asked.
#
if (parallel.newdata)
    ### UPDATED HERE, now using tch.tcnewdata. Added argument child.task
    newdata <- tch.tcnewdata (tbl = results, mat = leaf.matrix,
                              trees = big.list.of.trees, d.num = d.num,
                              child.task = parallel.newdata)

#
### UPDATED HERE### Removed clustering. This is a child task, we cannot cluster yet.
#

#
### UPDATED HERE### changed return values to what tch.combiner needs.
#
# Must return:
# -tbl
# -mat
# -extended.tbl
#
# Possibly return:
# -big.list.oftrees
# -dists, condition is parallel.dists
# -newdata, condition is parallel.newdata
#
# Removed:
# -final.clust, clustering not yet performed
# -d.num, redundant
# -final.algorithm, redundant
#
# Set up return value; add requested stuff.
#
return.val <- list(call = match.call(), tbl = results,
                   extended.tbl = original.results, mat = leaf.matrix,
                   additional.args = additional.args)

if (control$return.trees)
    return.val$trees <- big.list.of.trees
```

```
if (parallel.dists)
    return.val$dists <- dists
if (parallel.newdata)
    return.val$newdata <- newdata
class(return.val) <- "tch.child.treeClust"

return(return.val)
}#end function
```

(6) `tch.combiner()` Function Code

```
################################################################################
#
# Name:
# tch.combiner()
#
# Description:
# This function combines the output of all of the child tasks. If the child tasks
# did not compute distance and/or newdata, and we are required to do so, it is
# done now.
#
# The first child task's output is used to set-up all container objects, the
# remaining tasks are added to these objects in a loop. The results are then
# returned in the standard treeClust() function's objects.
#
# Arguments:
# -d.num: the tree-based dissimilarity measure to use
#
# -final.algorithm: the clustering algorithm to use, if specified by the user
#    at run time.
#
# -control: a treeClust.control() object
#
# -child.dir: the path to the child task output directory.
#
# -parallel.newdata: logical indicating if newdata was created by the child tasks.
#
# -parallel.dists: logical indicating if dissimilarity was computed by the child
#   tasks.
#
# -verbose: This argument is used for error checking, it forces function to
#   print to the screen so the user knows that the function call occurred
#
# Return:
# -A list containing three objects that hold the paths
#  to the three new directories
#
################################################################################
tch.combiner = function(d.num,
                        final.algorithm,
                                       control,
                                       child.dir,
                        parallel.newdata,
                        parallel.dists,
                        verbose = FALSE)
{

if(verbose == TRUE)
  cat('FUNCTION: tch.combiner \n')

#
# Get names of all of the child task output files. Some children may not have
# produced anything, because their tree's didn't produce anything
#
files.to.combine = list.files(child.dir)

#
# Because of the way that we save the files, their internal data frame always
# has the same name, so we can only load one at a time
#
# load the first file, and all objects
#

load(paste0(child.dir, '/', files.to.combine[1]))

#
# We always need to get tbl, extended.tbl, and mat from the child task output.
```

```
# We need the tress if haven't computed dissimilarity, and d.num = 3 or 4, or
# if we are asked to keep them. If the children computed dissimilarity/newdata,
# we need to get dists and/or newdata.
#
# The code for getting the first file is ugly, the rest is nice.
#

# Fist all of the objects that don't require dissimilarity computations
tbl          = out$tbl
extended.tbl = out$extended.tbl
mat          = out$mat

################################################################################
# The child tasks already checked the trees for size, so we don't do that here.
#
# We only need the trees if: 1) We're asked to keep them, 2) We haven't computed
# distances or newdata yet, and d.num = 3 or d.num = 4. or we else get rid of them,
# they're huge and take up space.
#
# Initially set big.list.of.trees to NULL so we can still run tcdist & tcnewdata
# with out error if we need to.
#
################################################################################
get.trees = FALSE
big.list.of.trees = NULL

if(control$return.trees){
       get.trees = TRUE
       big.list.of.trees = out$trees
}else if((parallel.dists == FALSE && control$return.dists == TRUE) &&
         (d.num == 3 || d.num == 4))
{
       get.trees = TRUE
       big.list.of.trees = out$trees
}else if((parallel.newdata == FALSE && control$return.newdata == TRUE) &&
         (d.num == 3 || d.num == 4))
{
       get.trees = TRUE
       big.list.of.trees = out$trees
}#end if

#
# Check to see if distance calculations were done in the child tasks, if so sum
# them. If not we will wait we have all of the child task outputs.
#

# distance
get.dists = FALSE
calc.dists = FALSE

if (parallel.dists == TRUE &&
    (control$return.dists == TRUE ||
     (!missing (final.algorithm) &&
      is.element (final.algorithm, c("pam," "agnes")))))
{
       get.dists = TRUE
       dists = out$dists
}else if (parallel.dists == FALSE &&
          (control$return.dists == TRUE ||
           (!missing (final.algorithm) &&
            is.element (final.algorithm, c("pam," "agnes")))))
{
       calc.dists = TRUE
}#end if

# newdata
get.newdata = FALSE
calc.newdata = FALSE
```

```
if (parallel.newdata == TRUE &&
    (control$return.newdata == TRUE ||
     (!missing (final.algorithm) &&
      is.element (final.algorithm, c("kmeans," "clara")))))
{
           get.newdata = TRUE
       newdata = out$newdata
}else if (parallel.newdata == FALSE &&
               (control$return.newdata == TRUE ||
           (!missing (final.algorithm) &&
            is.element (final.algorithm, c("kmeans," "clara")))))
{
       calc.newdata = TRUE
}#end if

#
# Now, loop through each child task output and combine them
#
for (i in 2:length(files.to.combine))
{

  load(paste0(child.dir, '/', files.to.combine[i]))

       #append new tree information to: tbl, extended.tbl, and mat
       tbl = rbind(tbl, out$tbl)
       extended.tbl = rbind(extended.tbl, out$extended.tbl)
       mat = cbind(mat, out$mat)

       #trees, if we need them
       if (get.trees)
              big.list.of.trees = c(big.list.of.trees, out$trees)

       #tcdist and tcnewdata
       if (get.dists)
              dists = dists + out$dists  #add the child dissimilarity matrices together

       if (get.newdata)
         newdata = cbind(newdata, out$newdata) #append the new leaves to the matrix

}#end loop

#
# Now we have the output from all of the trees so we can scale the distance and/or
# newdata object using the max tree quality, OR if the child tasks did not
# perform the computations we do it now.
#
# distance
if (get.dists && (d.num == 2 || d.num == 4))
  #divide the dissimilarity matrix by the max tree-quality
       dists = dists / (length(files.to.combine)*max(tbl[,"DevRat"]))
else if (calc.dists)
       dists = tch.tcdist(tbl = tbl, mat = mat, trees = big.list.of.trees,
                          d.num = d.num, child.proc=F)

# newdata
if (get.newdata && (d.num == 2 || d.num == 4))
  #divide the dissimilarity matrix by the max tree-quality
       newdata = newdata / (length(files.to.combine)*max(tbl[,"DevRat"]))
else if (calc.newdata)
       newdata = tch.tcnewdata(tbl = tbl, mat = mat, trees = big.list.of.trees,
                               d.num = d.num, child.proc=F)

#
# Set up return value; add requested stuff.
#

combiner.return =  list(call = match.call(), tbl = tbl, extended.tbl = extended.tbl,
                        mat = mat)
if (control$return.trees)
```

```
    combiner.return$trees = big.list.of.trees

if ((control$return.dists) ||
    (!missing (final.algorithm) &&
     is.element (final.algorithm, c("pam," "agnes"))))
    combiner.return$dists = dists

if ((control$return.newdata) ||
    (!missing (final.algorithm) &&
     is.element (final.algorithm, c("kmeans," "clara"))))
    combiner.return$newdata = newdata

class(combiner.return) = "tch.combiner"
return (combiner.return)

}#end function
```

(7) Updated `tcdists()` Function Code

```
##############################################################################
#
# Name:
# tch.tcdist()
#
# Description:
# This function is a modification of tcdists(), it allows child tasks to compute
# portions of dists - enabling parallel processing. The final portion of the
# dissimilarity computation occurs in tch.combiner().
#
# Change Highlights:
#  Changes are indicated by ###UPDATED HERE###
#
# List of Modifications:
# -For d2, and d4 removed scaling by the maximum tree-quality so that they can
#  be computed in parallel
#
##############################################################################
tch.tcdist <- function (obj, d.num = 1, tbl, mat, trees, verbose = 0, child.task = TRUE)
{

#
# Extract distances from an existing treeClust object. If there's
# a dist element in there, computed with the proper d.num, return it.
#
if (!missing (obj) && any(names (obj) == "dists") && obj$d.num == d.num)
    return (obj$dists)
if (!missing (obj) && any (names (obj) == "tbl"))
    tbl <- obj$tbl
else
    if (missing (tbl))
        if (d.num == 2 || d.num == 4)
            stop ("'Tbl' missing but required")
#
# For d1, use daisy() on the "mat" object; for d2, do the same
# but with weights.
#
# UPDATE: if child.task == TRUE the weights for d.num = 2 will
# now be just the DevRat, this allows for parallel processing.
# We divide by max DevRat in tch.combiner()
#
if (d.num == 1 || d.num == 2) {
    if (!missing (obj) && any (names (obj) == "mat"))
        mat <- obj$mat
    else
        if (missing (mat))
            stop ("For d1 or d2, this function requires the 'mat' element")
    if (d.num == 1) tree.wts <- rep (1, ncol(mat))
    if (d.num == 2)
      ###UPDATED HERE###
      if(child.task) tree.wts <- tbl[,"DevRat"]
      else tree.wts <- tbl[,"DevRat"] / max (tbl[,"DevRat"])
    return (daisy (mat, metric = "gower," weights = tree.wts))
}
#
# For d3 or d4 we need the trees.
#
# UPDATE: if child.task == TRUE the weights for d.num = 4 will
# now be just the DevRat, this allows for parallel processing.
# We divide by max DevRat in tch.combiner()
#
if (!missing (obj) && any (names (obj) == "trees"))
    trees <- obj$trees
else
    if (missing (trees))
        stop ("For d3 or d4, we need the trees")
```

```
n <- length (trees[[1]]$where)
dists <- numeric (n * (n - 1) / 2)
if (d.num == 3) tree.wts <- rep (1, length(trees))
if (d.num == 4)
  ###UPDATED HERE###
  if(child.task) tree.wts <- tbl[,"DevRat"]
  else tree.wts <- tbl[,"DevRat"] / max (tbl[,"DevRat"])

for (i in 1:length (trees)) {
    if (verbose > 0)
        cat ("Tree ," i, ," has wt ," tree.wts[i], "\n")
    dists <- dists + tree.wts[i] * d3.dist (trees[[i]])
}
class (dists) <- "dist"
attr (dists, "Size") <- n
attr (dists, "Diag") <- FALSE
attr (dists, "Upper") <- FALSE
attr (dists, "method") <- "manhattan"
return (dists)
}
```

(8) Updated `tcnewdata()` Function Code

```
################################################################################
#
# Name:
# tch.tcnewdata()
#
# Description:
# This function is a modification of tcnewdata(), it allows child tasks to compute
# portions of newdata - enabling parallel processing. The final portion of the
# newdata computation occurs in tch.combiner().
#
# Change Highlights:
#  Changes are indicated by ###UPDATED HERE###
#
# List of Modifications:
# -For d2, and d4 removed scaling by the maximum tree-quality so that they can
#  be computed in parallel
#
################################################################################
tch.tcnewdata <- function (obj, d.num = 1, tbl, mat, trees, child.task = TRUE)
{

#
# Compute the "newdata" data frame from a treeClust item. If there's
# a newdata element in there, computed with the proper d.num, return it.
#
if (!missing (obj) && any(names (obj) == "newdata") && obj$d.num == d.num)
    return (obj$newdata)
if (!missing (obj) && any (names (obj) == "tbl"))
    tbl <- obj$tbl
else
    if (missing (tbl))
        if (d.num == 2 || d.num == 4)
            stop ("'Tbl' missing but required")

if (!missing(obj) && any(names(obj) == "mat"))
    mat <- obj$mat
else if (missing(mat))
    stop("'Mat' misisng but required")

#
# We need to know the number of leaves in each tree (from which we subtract
# 1 to get the number of columns) and the sample size.
# Then we can set up newdata.
#
leaf.counts <- sapply(mat, function(x) length(unique(x)))
n <- nrow (mat)
start <- c(1, 1 + cumsum(leaf.counts[-length(leaf.counts)]))
end <- cumsum(leaf.counts)
newdata <- matrix(0, n, sum(leaf.counts))


#-----------------d1 & d2-------------------------------------
#
# For d1, use daisy() on the "mat" object; for d2, do the same
# but with weights.
#
if (d.num == 1 || d.num == 2) {
    if (!missing (obj) && any (names (obj) == "mat"))
        mat <- obj$mat
    else
        if (missing (mat))
            stop ("For d1 or d2, this function requires the 'mat' element")

    for (i in 1:length(leaf.counts)) {
        mod <- model.matrix(~factor(mat[, i]) -  1)
        if (d.num == 2)
```

```
        {
          ###UPDATED HERE###
          if(child.task)
            newdata[, start[i]:end[i]] <- mod * tbl[i, "DevRat"]
          else
            newdata[, start[i]:end[i]] <- mod * tbl[i, "DevRat"]/ max(tbl[, "DevRat"])
        }else newdata[, start[i]:end[i]] <- mod #assign the model
    }#end loop
    return (newdata)
}

#-----------------d3 & d4-------------------------------------
#
# For d3 or d4 we need the trees. Also we only produce (# leaves - 1)
# columns for each tree.
#
col.counts <- sapply(mat, function(x) length(unique(x)) - 1)
n <- nrow (mat)
start <- c(1, 1 + cumsum(col.counts[-length(col.counts)]))
end <- cumsum(col.counts)
newdata <- matrix(0, n, sum(col.counts))

#
# standard check for trees
#
if (!missing (obj) && any (names (obj) == "trees"))
    trees <- obj$trees
else
    if (missing (trees))
        stop ("For d3 or d4, we need the trees")

#
# Compute the matrix of pairwise leaf distances, then call
# cmdscale() on the result.
#
for (i in 1:length (trees)) {
    leaf.dists <- d3.dist (trees[[i]], return.pd=TRUE)
    newcols <- cmdscale (leaf.dists, ncol(leaf.dists) - 1)

    #
    # The row.names of "newcols" are the leaf numbers, not the numbers
    # found in the "where" element of the tree. So we convert...
    #
    w <- trees[[i]]$where
    r <- row.names (trees[[i]]$frame)[w]

    #
    # ...and then extract the correct rows of newcols.
    #
    if (d.num == 4)
      ###UPDATED HERE###
      if(child.task)
        newdata[, start[i]:end[i]] <- newcols[r,] * tbl[i,"DevRat"]
      else
        newdata[, start[i]:end[i]] <- newcols[r,] * tbl[i,"DevRat"]/max(tbl[, "DevRat"])
    else
      newdata[, start[i]:end[i]] <- newcols[r,]

}#end loop


return (newdata)
}#end function
```

(9) Updated `tch.treeClust.control()` Function Code

```
#############################################################################
#
# Name:
# -tch.treeClust.control()
#
# Description:
# This function is a modification of treeClust.control, it adds additional
# arguments used with tch.treeClust() and it's sub functions.
#
# Additional arguments:
# -MpC: Memory-per-Core. Median value of available RAM on the hamming supercomputer
#
# -BpO: Bytes-per-Observation. We assume the data is numeric, and most numbers
#       in R have a size of 8 bytes
#
# -Rub: Number of tasks used upper bound. Heuristic based on testing.
#
# -Rlb: Number of tasks used lower bound. Heuristic based on testing.
#
#############################################################################
tch.treeClust.control <- function (return.trees = FALSE,
                                   return.mat = TRUE,
                                   return.dists = FALSE,
                                   return.newdata = FALSE,
                                   cluster.only = FALSE,
                                   serule = 0,
                                   DevRatThreshold = 1,
                                   parallelnodes = 1,
                                   MpC = 3700,
                                   BpO = 8,
                                   Rub = 0.20,
                                   Rlb = 0.075,...)
{
  list(return.trees = return.trees,
       return.mat = return.mat,
       return.dists = return.dists,
       cluster.only = cluster.only,
       return.newdata = return.newdata,
       serule = serule,
       DevRatThreshold = DevRatThreshold,
       parallelnodes = parallelnodes,
       MpC = MpC,
       BpO = BpO,
       Rub = Rub,
       Rlb = Rlb, ...)
}#end function
```

APPENDIX B. FEASIBLE PARTITION TABLES

Table 7. Feasible Partition Table: Internet Ads Data

Split Number	Number of Tasks	Number of Variables per Task, x
1	1	1558
2	2	779
3	3	520
4	4	390
5	5	312
6	6	260
7	7	223
8	8	195
9	9	174
10	10	156
11	11	142
12	12	130
13	13	120
14	14	112
15	15	104
16	16	98
17	17	92
18	18	87
19	19	82
20	20	78
21	21	75
22	22	71
23	23	68
24	24	65
25	25	63
26	26	60
27	27	58

Split Number	Number of Tasks	Number of Variables per Task, x
28	28	56
29	29	54
30	30	52
31	31	51
32	32	49
33	33	48
34	34	46
35	35	45
36	36	44
37	37	43
38	38	41
39	39	40
40	40	39
41	41	38
42	43	37
43	44	36
44	45	35
45	46	34
46	48	33
47	49	32
48	51	31
49	52	30
50	54	29
51	56	28
52	58	27
53	60	26
54	63	25
55	65	24
56	68	23
57	71	22
58	75	21

Split Number	Number of Tasks	Number of Variables per Task, x
59	78	20
60	82	19
61	87	18
62	92	17
63	98	16
64	104	15
65	112	14
66	120	13
67	130	12
68	142	11
69	156	10
70	174	9
71	195	8
72	223	7
73	260	6
74	312	5
75	390	4
76	520	3
77	779	2
78	1558	1

Table 8. Feasible Partition Table: HAR Data

Split Number	Number of Processes	Number of Variables per Task, x
1	1	561
2	2	281
3	3	187
4	4	141
5	5	113
6	6	94
7	7	81

Split Number	Number of Processes	Number of Variables per Task, x
8	8	71
9	9	63
10	10	57
11	11	51
12	12	47
13	13	44
14	14	41
15	15	38
16	16	36
17	17	33
18	18	32
19	19	30
20	20	29
21	21	27
22	22	26
23	23	25
24	24	24
25	25	23
26	26	22
27	27	21
28	29	20
29	30	19
30	32	18
31	33	17
32	36	16
33	38	15
34	41	14
35	44	13
36	47	12
37	51	11
38	57	10

Split Number	Number of Processes	Number of Variables per Task, x
39	63	9
40	71	8
41	81	7
42	94	6
43	113	5
44	141	4
45	187	3
46	281	2
47	561	1

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. THE HAMMING SUPERCOMPUTER

Figure 28 shows a summary of the HSC's resources at the time of writing. A specific point of interest is the amount of RAM available per core, if all cores on a node are in use. The minimum amount available in this case is 3.7GB (3700MB), which is why our default *MpC* value is 3700MB.

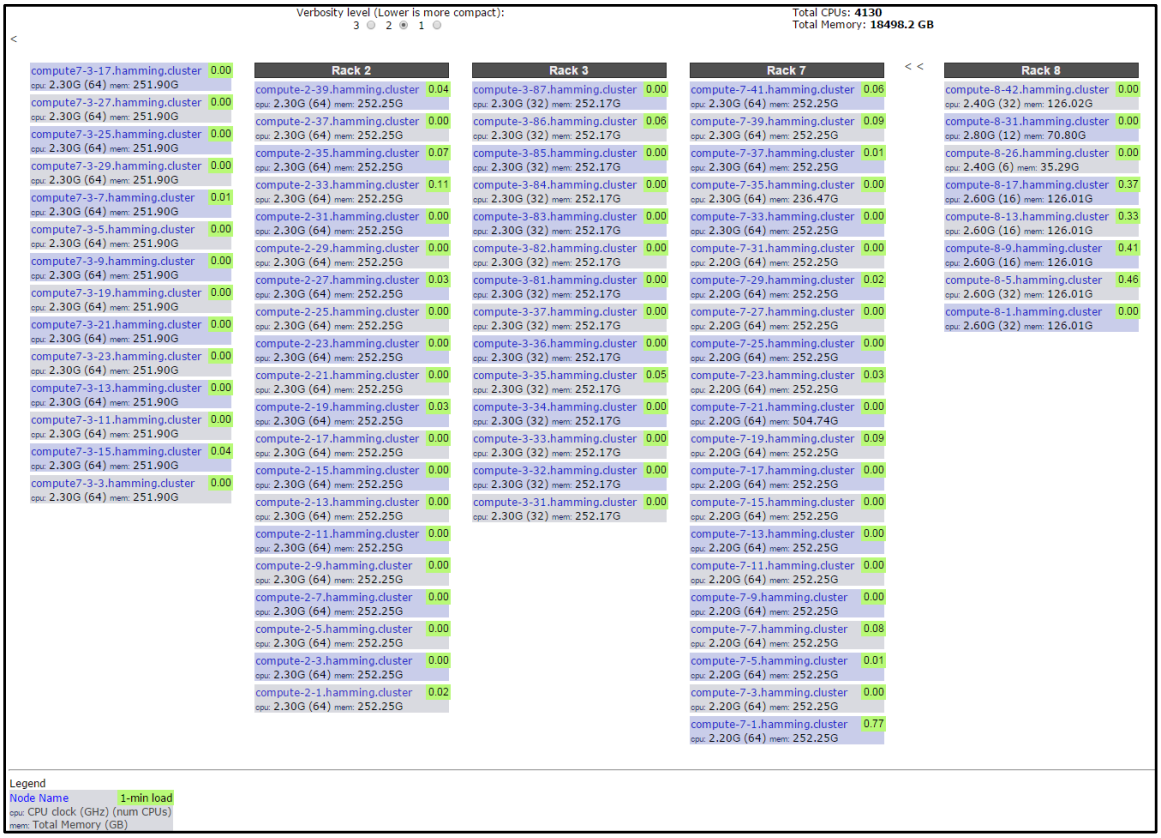


Figure 28. A Summary of the Hamming Supercomputer.
Source: Sharrock and Haferman (2017).

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Abd-El-Barr, M., & El-Rewini, H. (2005). *Fundamentals of computer organization and architecture.* Hoboken, NJ: John Wiley and Sons.

Almasi, G. S., & Gottlieb, A. (1989). *Highly parallel computing.* Redwood City, CA: The Benjamin/Cummings Publishing Co.

Amazon Web Services, Inc. (2017, May). *AWS batch*. Retrieved from https://aws.amazon.com/batch/

Anguita, D., Ghio, A., Oneto, L., Parra, X., & Reyes-Ortiz, J. L. (2013). A public domain dataset for human activity recognition using smartphones. *21st European Symposium on Artificial Neural Networks* (pp. 437–442). Bruges: ESANN.

Blair, S. (2012, April). General purpose graphical processing unit computing (GPGPU) at NPS. Monterey, California. Retrieved from https://calhoun.nps.edu/handle/10945/36595

Buttrey, S. E., & Whitaker, L. R. (2015). treeClust: An R package for tree-based clustering dissimilarities. *The R Journal, 7*(2), 227–236. Retrieved from https://journal.r-project.org/archive/2015/RJ-2015-032/index.html

Buttrey, S. E., & Whitaker, L. R. (2016). *A scale-independent, noise-resistant dissimilarity for tree-based clustering of mixed data.* Monterey, CA: Naval Postgradute School. Retrieved from http://hdl.handle.net/10945/48615

Eddelbuettel, D. (2017, February). *High-performance and parallel computing with R*. Retrieved from https://CRAN.R-project.org/view=HighPerformanceComputing

Fulton, B. M. (2016). *Determining market categorization of United States ZIP codes for purposes of Army recruiting (Masters thesis).* Monterey: Naval Postgraduate School. Retrieved from http://hdl.handle.net/10945/49463

Grolemund, G. (2014). *Hands-on programming with R.* Sebastopol, CA: O'Reilly Media.

Haferman, J. (2016). *HPC-HPCClusterTechnicalGuide-141016-1021-32*. Retrieved from NPS Wiki High Performance Computing: https://wiki.nps.edu/display/HPC/HPC-HPCClusterTechnicalGuide-141016-1021-32#HPC-HPCClusterTechnicalGuide-141016-1021-32-_Toc465681838

Haferman, J. (2017, Feburary). *Hamming architecture*. Retrieved from NPS Wiki High Performance Computing: https://wiki.nps.edu/display/HPC/Hamming+Architecture

Harris, D. M., & Harris, S. L. (2007). *Digital design and computer architecture.* San Francisco: Elsevier Inc.

Hartigan, J. (1975). *Clustering algorithms.* New York, NY: John Wiley and Sons.

Horstmann, C. (2008). *Big java* (3rd ed.). Hoboken, NJ: John Wiley and Sons, Inc.

Kurose, J. F., & Ross, K. W. (2010). *Computer networking a top down approach* (5th ed.). New York, NY: Addison-Wesley.

Kushmerick, N. (1998, July). *Internet Advertisements Data Set*. Retrieved from UCI Machine Learning Repository: https://archie.ics.uci.edu/ml/datasets/Internet+advertisements

Lichman, M. (2013a). *Human Activity Using Smart Phones Data Set*. Retrieved from UCI Machine Learning Repository: http://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones

Lichman, M. (2013b). *Internet Advertisements Data Set*. Retrieved from UCI Machine Learning Repository: https://archive.ics.uci.edu/ml/datasets/internet+advertisements

Lockwood, G. K. (2017, March). *lapply-based parallelism*. Retrieved from Data-Inensive Computing: www.gleenklockwood.com

Lynch, S. K. (2014). *A scale-independent clustering method with automatic variable selection based on trees* (Master's thesis). Monterey: Naval Postgraduate School. Retrieved from http://hdl.handle.net/10945/41412

Maechler, M., Rousseeuw, P., Struyf, A., Hubert, M., Hornik, K., Studer, M., . . . Gonzalez, J. (2017). Cluster: Methods for cluster analysis. Retrieved from https://CRAN.R-project.org/package=cluster

Marsh, B. D., LeBlanc, T. J., Markatos, E. P., & Scott, M. L. (1991). First-class user-level threads. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles,* (pp. 110–121). Pacific Grove, CA. doi:10.1145/121133.344329

Miller, J., & Apatow, J. (Directors). (2006). *Talledega nights: the ballad of Ricky Bobby* [Motion Picture].

Patterson, D. A., & Hennessy, J. L. (2007). *Computer organization and design* (3rd ed.). Burlington, MA: Elsevier.

R Core Team. (2017, March). R: a language and environment for statistical computing. Vienna, Austria: R Foundation for Statistical Computing. Retrieved from https://www.R-project.org

Sharrock, N., & Haferman, J. (2017, April). *Ganglia (Hamming system monitor and management)*. Retrieved from NPS High Performance Computing: https://hamming.uc.nps.edu/ganglia/

Shiva, S. G. (2008). *Computer organization, deisgn, and architecture* (4th ed.). Boca Raton, FL: Taylor and Francis Group, LLC.

SLURM Workload Manager. (2017, April). *Documentation*. Retrieved from SchedMD: https://slurm.schedmd.com/documentation.html

Tanenbaum, A. S. (2008). *Modern operating systems* (3rd ed.). Upper Saddle River, NJ: Pearson Education.

Therneau, T., Atkinson, B., & Ripley, B. (2015). Rpart: Recursive partitioning and regression trees. *R package version 4.1-10*. Retrieved from https://CRAN.R-project.org/package=rpart

Tierney, L., Rossini, A. J., Li, N., & Sevcikova, H. (2016). now: Support for simple parallel computing in R. Retrieved from https://CRAN.R-project.org/package=snow

van Engelen, R. (2017, March). Algorithms part 1: Embarrassingly parallel. *High Performance Computing*. Tallahassee, FL: Florida State University. Retrieved from https://www.cs.fsu.edu/~engelen/courses/HPC/

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California